Developing A Virtual Modular Synthesizer For Sound Waves And Midi

Margaret Jagger
The College of Wooster, mjagger22@wooster.edu

Developing a Virtual Modular Synthesizer for Sound Waves and MIDI

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Bachelor of Arts in Computer Science in the Department of Mathematical & Computational Sciences at The College of Wooster

by
Margaret Jagger
The College of Wooster
2022

Advised by:

Drew Guarnera (Mathematical & Computational Sciences)

Abstract

Modular synthesis involves the alteration and modification of digital sound signals. Thus, this modular synthesizer allows a user the option of supplying their own MIDI-compatible controller to serve as an input source, or to use the built-in pure sound waves instead. Either input will be fed into the domain-specific language *SuperCollider* and altered, with specific sound modifications dependent on the input source used. Using theoretical knowledge of the physics behind the motion of sound waves, various *modules* and functionalities are created. Then, with *SuperCollider*, these modules are implemented into a synthesizer which accepts either pure sound waves or MIDI as inputs, in a clean and easy-to-use interface. This modular synthesizer also has the potential to be continually expanded and improved upon beyond this initial version, with the option of adding other functionalities and sound modification options to augment the features currently available.

Acknowledgments

First, the success of this project would not have been possible without my advisor, Professor Drew Guarnera, and his incredible patience and guidance throughout the IS process. His advice and support has made this an enjoyable experience, and less scary and stressful than I had been led to believe. I would also like to thank my academic advisor, Dr. Nathan Sommer, and the rest of the Wooster Math and CS Department for the suggestions and resources for the successful completion of this project. Also, to Dr. Heather Guarnera and Dr. Sofia Visa, I am forever thankful for your knowledge and contributions to my growth your classes have provided me. Thank you to Tessa from the Writing Center for answering my many questions about niche music topics, and her rationality in perspective while helping me to de-stress. Thank you to Sam, Ben, and Hawi for helping me to keep my days lively and upbeat while in the depth of my writing. To Yvonne and Dr. Niklas Manz, thank you for teaching my least stressful (and possibly, my favorite) class during my last year at Wooster. Additionally, I would like to thank my family for their support, both in and out of Wooster, allowing me to attain the success that I have had both inside and outside the classroom. Without them, much of this would not have been possible. Last, and certainly not least, thank you to every friend I have had, and all the fantastic people I have met along the way. To Andrea, Stacey, Libby, Jenna, and Katie, thank you for the late nights, and the fun moments when they were needed most. The support of each of these people have been incredibly helpful and uplifting while undertaking the struggles that are part of the IS process, and I am extremely grateful to each of them.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

Within the world of digital signal processing (DSP), digital audio, and physical instruments, there are almost limitless possibilities available now to create sounds through computers, whether through synthesis, production, or mixing. Ultimately, the products which result from explorations between computer science and music are dependent on the creator's familiarity with both fields. Current applications for music synthesis are sufficient in usability, with Helm, an open-source modular synthesizer, as one example. One is easily able to find Helm through an Internet search, download the application, and begin altering an input's sound to their liking, provided they have a MIDI controller on hand to serve as the input device. Thus, the primary goal of this thesis is to create a modular synthesizer with two options for inputs: MIDI and pure sound waveforms, such as sine waves.

There are three goals that must be accomplished in order to provide a modular synthesizer which utilizes both MIDI and pure sound waveforms: synthesizing (creating) pure sound waveforms through our language of choice *SuperCollider*, accepting MIDI as an input option, and altering either input properly so the sound modification is obvious. So, this project relies on knowledge of both digital signals (the detectable digital impulses through which messages or other information can be transmitted [38]), for MIDI input, as well as waveforms types, and mathematically, the ways in which we are able to alter these signals. Creating a virtual modular synthesizer requires a significant amount of programming in the domain-specific language SuperCollider, which connects the input from a physical MIDI controller and pure sound waveforms to the desired sound modifications. Thus, there will be a front-end graphical user interface, as well as a back-end which is connected to the signal modification modules. The front-end will provide users with a way to easily interpret and understand the modules of this synthesizer, and the back-end will process the desired sound changes the user has input, and output these alterations. The back-end includes modules typically found in open-source and commercial modular synthesizers, including, but not limited to: volume control, pitch bend, harmonics layering, delay, legato and staccato, distortion, and manual MIDI adjustments. This virtual modular synthesizer successfully completes each of these goals, and will serve as the primary product of this thesis.

Several of these modules, or functionalities which are able to act independently from each other, are self-explanatory; the pitch bend will alter the perceived pitch (frequency) of the input note, and the volume control will adjust the sound's loudness. The other modules mentioned are less intuitive. Harmonics layering involves layering various frequencies, whether the same or slightly different, over one another. Delay affects how early or late a note is perceived to be played. Legato, as defined in music, is the playback of notes in a connected manner, and staccato is the opposite, the playback of notes in a detached and separated way. Distortion typically is a *destructive* effect, in which the sound to be distorted cannot be reverted back to its previous state. For this modular synthesizer, distortion will act as another version of harmonics layering, adding unneeded and unpleasant sounds to a user's input. For MIDI input, there is also an option to manually adjust the sound output a user receives, through adjusting an *ADSR envelope*, or the Attack, Decay, Sustain, and Release of a MIDI note.

First, we will provide an introduction to the field of DSP, sound synthesis, and

MIDI, and SuperCollider, the domain-specific programming language used for this project. This discussion will include context on previous developments on modular synthesizers and define the difference between modular sound synthesis and other types of synthesis. The modular synthesizer, specifically, is a type of synthesizer, or electronic instrument, which is able to produce a wide variety of sounds, through sound modifications, in a single unit with a unified control system. Then, we discuss the important mathematical details which are relevant to the creation of this modular synthesizer. Specifically, this will include the various types of waveforms, and the ways in which the waves can be manipulated. After, we go into detail of the process of developing the modular synthesizer, and the step-by-step description of the ways in which each module was developed. Some modules proved to be more challenging to implement than others, and so we also provide context as to how a particular module is developed. Finally, we finish with a conclusion discussing the challenges of implementing this synthesizer, the ways in which this synthesizer could be expanded or improved, and suggestions for future work.

CHAPTER 2

Background

2.1 Related Works

The 1950s was a turning point within the field of electronic music and communica-tional technologies. This marked the creation of electronic music, with the use of physical components. Electronic music was rare, with synthesizers and electronic

Figure 2.1: The Minimoog Model D
[8]

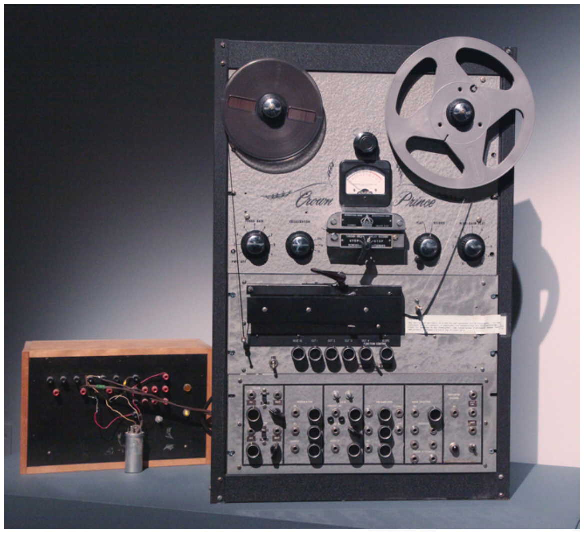

Figure 2.2: The Bode Audio System Synthesizer
[22]

instruments (examples of which include the electric guitar, electric keyboard, electric bass guitar, and more) being less common. Modular synthesizers were also rare, at that point they contained fewer modules than found in modular synthesizers today. These modular synthesizers were large, taking up significant space, as evidenced by Bode's Audio System Synthesizer (Figure 2.2) [8] [40] and Moog's Minimoog Model D (Figure 2.1 [22]) [33] [35], two influential analog modular synthesizers.

The first module, or functionality, created was the positive feedback oscillator, sometime between 1912 and 1914 [15], a module now commonly found in modular synthesizers, both in analog (physical) and digital (virtual) synthesizers. A positive feedback oscillator–or as it is known today, the oscillator–produces sound in which the output varies up and down (it oscillates) in a repeating pattern [29]. This is where the oscillator module gets its name, producing a sound which rises and falls, or oscillates. The resulting sound an oscillator produces will typically be one of four fundamental waveforms: the sine wave, triangle wave, sawtooth wave, square wave, or less commonly, the pulse wave. The creation of this positive feedback module is attributed to Austrian engineer and physicist Alexander Meissner. Unlike the oscillator module found today, the early version was only able to last a few minutes. Instead of continuously able to produce an audio waveform, this version produced only a small output of power [12]. 1912 introduced the communicational technologies field as something more than niche. Products from proper physics laboratories were rare to find, and so were machines capable of producing audio waveforms with high frequencies after several minutes. Meissner created a new type of machine: a high-frequency generator from an amplifier, with a vacuum tube as a substitute for a high-frequency receiver machine [12]. Previous to Meissner's creation, it was impossible to establish a connection between a high-frequency machine with a receiver. The output frequency of such a machine was not the same as the user's desired output frequency, a synchronous and superimposed frequency. Meissner's product was easier to handle than these machines, and much more useful. In addition, Meissner's machine was much less noisy as well. This machine could change the output frequency given to the receiver, and tune the receiver machine to a range of wavelengths that were previously not possible [12]. This development changed the process of receiving wavelengths, and resulted in the un-dampened wave. An un-dampened audio wave is the typical audio wave which oscillates without any force or opposing motion to prevent it from oscillating. This differs from a dampened wave, which is an audio wave which oscillates, yet there is some force or opposing motion which slows the wave, or stops it completely. Eventually, the dampened wave will lose energy, such that it will stop oscillating, unless there is an external force or motion to cause the wave to continue to oscillate. This un-dampened audio wave which resulted allowed for a user to fine-tune the frequency, while also suppressing any atmospheric distractions or sounds. With Meissner's work, the field's knowledge of constant sound generation and high

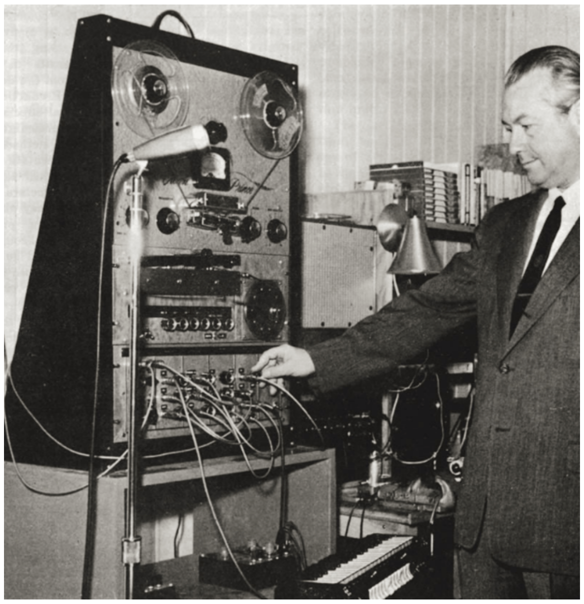

Figure 2.3: Harold Bode demonstrating the Bode Audio System Synthesizer

frequencies was expanded, and paved the way for future work on the short-wave band.[1]

Then, sometime between 1959 and 1960, Harold Bode (1909-1987), a German engineer, created the modular synthesizer which we are familiar with today. Bode was a German engineer and designer of audio tools. He foresaw that transistor technology (a semiconductor device which is used to amplify, control, and generate electrical signals) would become a key change in creating and designing synthesizers, especially modular synthesizers [15]. Transistors would link the audio signal through cables, through where each component, or "module" could be connected in any order, according to the user's preferences. Multiple modules–including modulators, filters, reverberation generators, and more–could then be connected in any order, either to modify or generate sounds. With transistor technology, Bode created his *Audio System Synthesizer* (Figure 2.3 [9]), which allowed for a larger

[1]Though not discussed in this paper, the short-wave band is typically used in maritime communications, international, and radio broadcasting. More reading on Alexander Meissner and his patented solution can be found in the Engineer and Technology History Wiki's page on Alexander Meissner here: `https://ethw.org/Alexander_Meissner`

number of sound creation possibilities than before. The system itself contained inputs for various sound sources, and the input signals could be modified by filter modules and a modulator [4]. As the system was modular, there were independently working modules for sound modification. These modules could be combined with each other in several ways, according to the user's desired order. This Audio System Synthesizer also made an impression on Robert Moog, who would take Bode's idea and further develop it [15].

Robert Moog (1934-2005) was an American engineer inspired by Harold Bode's Audio System Synthesizer. As the inventor of the first commercial synthesizer, dubbed the Moog synthesizer, Moog created the first integrated synthesizer (a synthesizer in which the modules are integrated into the system itself, rather than needing to be connected through cables). The Moog synthesizer was built in 1964, and contained several of the fundamental synthesizer concepts found today [35]. These modules on the Moog synthesizer included the voltage-controlled oscillator, voltage-controlled filter (a module which reduces or outright removes certain frequencies and harmonics from sound that is passed through [29]), envelope generators (a module which is used to shape the volume, or dynamics of a note when connected to an amplifier module, as well as alter the frequency or timbre of a note if connected to a filter module [29]), and the pitch wheel. With the Moog synthesizer, synthesizers were brought to a wider audience, influencing the development of popular music [35]. What followed the Moog synthesizer was the Minimoog Model D (Figure 2.1 [22]), a portable 1970 creation. The Minimoog Model D is acknowledged to be the most "influence synthesizer of all time" [15] due to its playability and compactness. Similar to the Moog synthesizer, it included modules to both generate and alter sounds [34].

The 1960s also saw the introduction of Donald "Don" Buchla, and his Buchla 100 Series Modular Electric Music System. Don Buchla (1937-2016) was another

American inventor, working in the field of sound synthesis independently of Robert Moog. The Buchla 100 Series (otherwise known as the "Buchla Series 100" or "Buchla Box"), is a keyboard-less modular synthesizer. Unlike Moog's Moog synthesizer and Minimoog Model D, the Buchla 100 Series used a more manual control [33]. Instead of a keyboard for user input, the user would control the synthesizer's modules directly. It contained a logical layout, with an intuitive, user-facing front panel, which allows the user to directly patch and route modules together, using patch cords. With the various knobs and triggers on the synthesizer itself, a user could manipulate sound according to multiple parameters. Along with user input differences, there were other important distinctions between the Buchla and Moog synthesizers. These changes include Buchla's use of voltage-controlled oscillators with multiple complex modulation options, and a decrease in emphasis on using filters. This methodology would lead to a division in school of thought: the East Coast school of thought, and the West Coast school of thought [15]. The West Coast method, which was led by Buchla, put greater emphasis on the experimental side of electronic music, and so designed their synthesizers according to this.

Both the East Coast and West Coast methods of audio synthesis are important, as they eventually become the two general methods for synthesis: additive synthesis and subtractive synthesis (two other types of audio synthesis, to be explained in section 2.2). In a general overview, the East Coast approach embodies a straightforward and practical approach, with efficiency and reliability as important concepts, while West Coast synthesis focuses on experimentation and the inclusion of non-tradition modular controllers such as the touchpad [29]. For East Coast audio synthesis, its patches and way of forming modules fall under the category of "subtractive synthesis". Subtractive synthesis involves starting with a complex audio waveform, with multiple harmonics (i.e. pitches, or notes) [47]. Then, a low-pass filter (or in analog synthesizers, a voltage-controlled low-pass filter, or VCF) slowly removes selected frequencies from the complex wave. The sound produced by West Coast synthesis is less known that East Coast synthesis. In place of aspiring towards a sense of musical efficiency, the West Coast method evolved out of the desire to replicate the acoustically-generated tones through manipulating previously recorded sounds [31]. Thus, the West Coast method is not centered around a singular principle. It combines elements of additive synthesis (the addition of multiple, simple audio waveforms together to create a complex, composite waveform–often sine waves) [31] and frequency modulation to create many sounds of complex timbres. Both additive and subtractive synthesis will be discussed further in the next section.

2.2 Modular Synthesis: What Is It?

A modular synthesizer is a type of synthesizer which is composed of separate modules for different functions. These modules, in a physical hardware synthesizer, would be connected by the user to create a "patch." The output from these modules would then be audio signals, voltages, or digital signals for various logical or timing conditions. Typically, these modules would include voltage-controlled oscillators, voltage-controlled filters, voltage-controlled amplifiers, and envelope generators. There was a need for voltage-controlled modules, such that the module would not receive sufficient power otherwise in order to properly function. Within a virtual modular synthesizer, we have no need for these voltage-controlled modules. However, we still must "patch" these modules together in a linear sort of way, and decide the order in which the modules are applied to a digital signal.

To do so, there is no set order. Virtual modules, unlike their physical counterparts, can easily be patched together in any order. Based on the order that the user changes the values of each module, the synthesizer itself will then change the audio output to align. It is user dependent on which module is triggered first.

Various types of modules are used within synthesizers, such as oscillators, filters, amplifiers, mixers, envelope generators, sequencers, and much more. The basic modular functions involve effects to an audio wave's signal itself, the control of an audio wave, or the logic–or timing–of the audio signal [15]. These modular functions can be categorized into one of two groups: a source module, or a processor module. A source module can be characterized by a certain out, but has no signal input value. There are multiple types of source modules, the main one being the oscillator (with the primary ones being either a voltage-controlled oscillator–VCO–or a low-frequency oscillator–LFO) which impact the sound that is generated from a synthesizer. The oscillator itself is a type of module which generates a repeating signal [15]. The rise and fall of the signal is what gives the module its name, as the signal will oscillate between its trough and its peak [38]. The speed with which the wave rises and falls, or reaches its peaks and troughs, is known as the wave's *frequency*. If the frequency is not too low or too high, then the human ear will be able to hear it. The generally accepted range of human hearing is between 20 Hz and 20 kHz, and so the frequency of audio waves will fall between these two values. The VCO will output a signal, where the signal's frequency is a function of the oscillator. In its most basic form, a VCO will be a simple waveform, usually a square or a sawtooth wave, which can be changed dynamically through a control such as frequency modulation. The LFO will be operated using a period with a length of anywhere from a fortieth of a second up to several minutes. Generally, the LFO will be used as a control for a different module; with an oscillator, the LFO will produce a modulation to the signal's frequency, and with an amplifier, the LFO will produce a change in the signal's amplitude. Certain changes to a signal's frequency may produce an effect known as *vibrato*. Vibrato is a musical effect in which there is a small, regular, and pulsating change to a signal's pitch. It is most used to add expression and phrasing to both vocal and instrumental music, and is characterized by two factors: the first, the amount the pitch is varied (the "extent of vibrato"), and the second, the speed at which the pitch is varied (or the "rate of vibrato"). Modulations to the signal's amplitude are different. These changes may produce *tremolo*, or a trembling effect in music. There are two distinct types of tremolo: rapid reiteration, and one based on a variation in a signal's amplitude. Rapid reiteration will occur due to either a single note, two notes, two chords, or as a roll on a percussive instrument. As a single note, it is particularly noticeable on a bowed string instrument, such as the violin or viola. On these instruments, a single note may be played by rapidly moving the bow back and forth. On a non-bowed instrument, such as the guitar, tremolo can be produced by plucking a single note repeatedly. As tremolo between notes or chords, these will be played in alternation, switching between note or chord one, then quickly playing note or chord two, and back again. At times, the terms for tremolo and vibrato in non-classical music are used incorrectly or interchangeably. In the world of classical music, they are both properly defined as separate musical effects. Vibrato is defined as the periodic variation in a note or signals' pitch (frequency), while tremolo is defined as the fast repetition or the same note or signal to produce the effect of a longer note being sounded. In practice, however, it is difficult for a performer to achieve pure vibrato or pure tremolo, as only a note's pitch (frequency) or volume (amplitude) would be allowed to be varied. So, variations to both pitch and volume will often be made at the same time. In the world of popular music, this means that the definitions for both terms will be modified slightly; vibrato will retain its same meaning as it does in classical music, with a periodic variation in a note or signal's pitch, but tremolo will instead refer to a periodic variation in volume, achieved by other electric effects.

Processor modules are different, and characterized by having both a signal input and an output. These modules include the filter, amplifier, low-pass gate, mixers, limiters, and more. The first type, the filter, in its most basic sense will remove (or filter out) frequencies from an audio signal, as desired by the user. The most common types of filters are the low-pass filter (LP) and high-pass filter (HP). Both, as evidenced by their name, will filter out frequencies of a certain band, and let other frequencies pass. Low-pass filters will filter out high frequencies, allowing low frequencies to pass, while high-pass frequencies will filter out low frequencies, letting high frequencies pass [47]. Other less common filter types include the bandpass (BP) filter, and all-pass filters, on certain synthesizers. The bandpass filter will attenuate both low and high frequencies, allowing only a certain range of frequencies around a specified cutoff point to pass through. The point at which a filter will begin working is known as the cutoff point, which is set by the module's cutoff knob itself. Filters will be used to alter the timbre, or the tone of the signal's sound. Another type of processor module is the amplifier. An amplifier is a component of a modular synthesizer which will change the amount of a signal which passes through the module.

There are four types of synthesis: modular synthesis, what this paper will focus on, wavetable synthesis, additive synthesis, and subtractive synthesis. A wavetable is a collection of single-cycle waveforms, or samples of audio, which are played on a loop to produce a periodic waveform, such as a sine wave. Then, wavetable synthesis will use a table, which contains the values of several frequencies played in a certain order (the wavetable). When a note is pressed, or a MIDI note on command is given, the signal will move through the table in the order specified, smoothly changing the shape of the signal into the various waves specified on the table. This produces an output which can evolve both quickly and smoothly, as it will follow the table of waveforms to modify the input signal. It will typically offer the widest range of sound that could be created or modified, in comparison to other techniques of sound synthesis [15].

Subtractive synthesis begins with a complex, composite audio wave. The input signal is stripped of its extra sounds and layers, as it goes through a chain of modules. Some harmonics present in the input signal will be reduced into harmonic structures which mimic the harmonics of actual instruments, or will be reduced to a fundamental harmonic sound, into its base waveform [15]. The modules associated with subtractive synthesis involve filters and envelop adjustments. The filter's "cutoff frequency" or "cutoff point" changes depending on the type of filter being used. The main filter types used in subtractive synthesis include the low-pass filter, the high-pass filter, the band-pass filter, and the notch, or band-reject filter. The two new filter types to us are the band-pass filter, and the notch filter. The band-pass filter will only let a select range of frequencies pass through it, with its cutoff frequency as the center point for this range. The notch filter will do the opposite, removing a select range of frequencies around a certain cutoff point [15].

Additive synthesis is the opposite of subtractive synthesis. We begin with a rudimentary audio wave or waveform, typically a sine wave. Then, sounds, frequencies, or harmonics are added on top of this fundamental audio waveform, creating a new timbre [47]. It attempts to achieve the same goal as subtractive synthesis, only through constructive measures, rather than destructive.

This project will focus on including several distinct modules for this prototyped modular synthesizer: a pitch bender, volume control, harmonics layering, legato and staccato buttons, distortion, and manual adjustments. Each of these modules will be a processor-type module. They will contain a signal input (the MIDI note a user plays, or pure sound waves), and output modulations to this note, based on the module selected. The order in which these modules are used is entirely user-dependent, and does not need to fall into a set linear order.

2.3 MIDI

2.3.1 What is MIDI?

The Musical Instrument Digital Interface (more commonly known as MIDI) is a digital communications protocol which allows for multiple hardware and software electronic instruments, controllers, computers, and related devices to communicate over a connected network [18]. It is used most to translate performance "events" (or musical notes) into its equivalent digital message, and then transmit these messages to other MIDI devices. These devices (MIDI receivers) can control sound generators or performance generators to create or modify music. Any MIDI-compatible device can send or receive MIDI messages from a MIDI controller (or the device sending the MIDI messages), and this will include all types of synthesizers. As an "interface," MIDI is composed of a data communications link, and a system of hardware and software connected through this MIDI network. With MIDI, any electronic instruments and devices which are within a network can be worked with, through the transmission of a real-time performance and MIDI messages. These transmissions of performances and messages can then be put through the system to various instruments and devices through one singular data line, rather than multiple data streams, as this system can be chained from one device to another. A single data cable used with MIDI is capable of transmitting a real-time performance and MIDI-control message over 16 distinct channels, numbered appropriately one through sixteen. The musician working with the system will determine which of these channels to send information through, depending on which MIDI devices are being used [37].

However, there are several limitations to the MIDI protocol. The first and most important limitation of MIDI is that it does not support sound. It is unable to communicate audio itself, or create sounds [18]. Instead, as a digital language, it

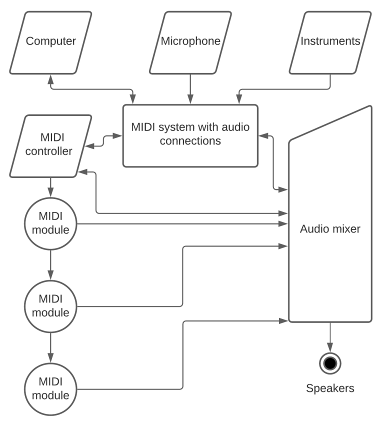

Figure 2.4: The MIDI system, with audio connections

instructs a compatible device or program to create, playback, or modify sounds. It will communicate an on/off status of a sound trigger, along with a range of parameters which instructs a MIDI receiver to control specified audio-related functions [21]. So, the data pathways for MIDI and audio routing will be different, even if they share a physical transmission cable, as in Figure 2.4 [18].

Additionally, much of MIDI is built around the concept of keyboard notes and pitches. MIDI messages are primarily transmitted through the use of an electronic keyboard. So, for other types of MIDI instruments (such as a violin, or clarinet), there is a restriction to how a note may sound using MIDI. Certain characteristics of non-keyboard instruments (such as the ability of playing discrete semitone pitches[2]) are more easily lost. For players of acoustic instruments, these issues are even more clear. Within MIDI, the velocity is considered to be a single note-on velocity, defining the dynamic response of the note to one value. For players of acoustic instruments, the velocity, or dynamic response, of a singular note is shaped by the player, along with the note's timbre and pitch when played [21].

[2]A discrete semitone pitch is another way of describing one pitch of the twelve-octave equal temperament tuning system.

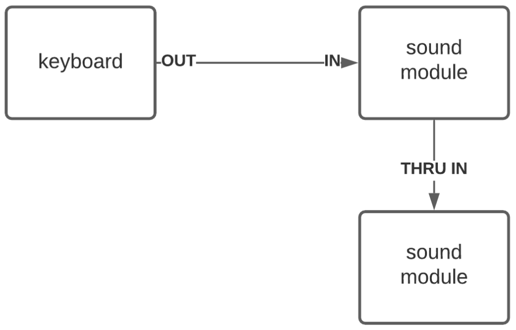

Figure 2.5: MIDI sockets using an electronic keyboard

2.3.2 How does MIDI work?

From a hardware perspective, the MIDI protocol will determine which types of plugs can be used for MIDI connections. There are three possible "sockets" that can be used on any MIDI-compatible device:

1. MIDI OUT: this will send data to other devices (MIDI receivers). An example of this will include an electronic keyboard which plays a note, and then "note messages" are sent out from the MIDI OUT socket.

2. MIDI IN: this socket will receive the MIDI information from other devices. Using the previous example, if a keyboard's MIDI OUT socket is connected with a MIDI cable to another sound module's IN socket, then the sound module will be able to produce sound on behalf of the keyboard.

3. MIDI THRU: this socket will relay the messages received at the MIDI IN socket, so more devices will be able to be chained together.

Typically, this will result in the flow in Figure 2.5, as it is normal for a keyboard's OUT port to be connected to a sound module's IN port. This IN socket will then be connected to another sound module, through the THRU IN port of the first sound module, into the IN port of the second sound module. Thus, both modules are now driven by the keyboard [21].

In a software view of the protocol, there are two basic types of messages MIDI can send: a *channel* message, and a *system* message. Channel messages are much more common. As previously mentioned, MIDI allows for the use of up to 16 different sounds to be controlled at the same time, through the use of its 16 different channels. So, each sound that must be played concurrently will be placed into a different *MIDI channel*. Within the channel, there are seven distinct messages, each of which contain a specific meaning and role. The first is the "Note on" message. This is the most common MIDI message used, and is sent whenever a note on a MIDI controller is played or pressed down, most common as pressing a key on a keyboard. The data contained in this message will tell a sound module how hard and fast a note was played (the *velocity* of a note), as well as which channel the note was played on [37]. The pitch of the message represents which key was pressed, and is defined as the number for each semitone on a keyboard. The semitone is the smallest interval of the modern Western tonal system. In equal temperament[3]–a tuning of the scale, based on a cycle of 12 identical fifths, and with the octave divided into 12 equal semitones–a semitone is the $\frac{1}{12}$ part of an octave. The notational system (i.e. sheet music) allows for three types of semitones to be distinguished: the diatonic, or the minor second (e.g. E-F, or C♯-D), the chromatic, the difference between a major second and a minor second (e.g. F-F♯, or D♭-D), and the enharmonic, the doubly diminished third (e.g. G♭♭-E) [11]. Middle C (C_4) is number 60 on the keyboard, and each semitone above and below Middle C is incremented or decremented accordingly [21].

The second is the "Note off" message, which turns off a note, or notes when a note is stopped. Like with Note on, Note off also contains values for channel, pitch, and velocity. For this message, only velocity's definition is changed, referring

[3]Equal temperament is regarded to be the normal tuning of the West, a 12-note chromatic scale. This is also known as A440, in which the note A_4 is tuned relative to the standard pitch of 440 Hz, and other notes are tuned as a specific number of semitones from this pitch.

instead to the speed at which a note is released. The final common type of channel MIDI message is the "Pitch-bend," which for electronic keyboard appears in the form of a physical wheel or sideways-moving handle of some type. Like its name implies, users will use this wheel or module to bend the pitch of notes currently being played. If the module is a physical wheel, it will return to its default position of zero (not bending a note) when the player lets go of the wheel. The pitch-bend will contain two pieces of data: the channel, and the pitch bend value. This value ranges from 0-127, with a value of 0 representing that the pitch has a full downwards bend to it, while a value of 127 represents the opposite, with a full upwards bend [21]. For this module, the value 64 will roughly indicate the center position on the wheel, and represent that a pitch has no bend applied to it.

MIDI also contains "system" messages, sent to all devices within the MIDI system [37]. These messages are not limited to a specified number of channels, and allow for a greater variety of data to be sent. There are three basic types of system messages [21]:

1. real-time: this type of message allows the devices within the MIDI system to synchronize together.

2. common: this message allows the devices to agree on some type of common musical issues, with tuning and song selection as two examples.

3. exclusive: this message will exclusively send data to one device type. For manufacturers, these messages are used to send data to only one type of synthesizer or device type, serving as "add-ons" to MIDI.

System messages are much harder to understand than channel messages, and so will not be discussed in great detail within the scope of this project.

2.3.3 Constructing MIDI Messages

MIDI messages are composed of three bytes, with a byte defined as a fundamental computation building block which consists of eight binary digits (or bits), either zero or one. Bytes which begin with a zero will be a "data byte" and bytes which begin with a one are "status bytes." The most significant bit (the MSB, or the leftmost binary bit within a MIDI message) identifies the byte type. As mentioned, bytes which have an MSB of 0 (or which begin with a 0) are data bytes, while bytes with an MSB of 1 are status bytes.

a status byte: *1xxx nnnn*

a data byte: *0nnn nnnn*

A status byte determines the type of MIDI function that will be performed, and encodes channel data, allowing the instruction to be received by a device that is set to respond to the selected channel. A data byte is used to associate a certain value to the event that is given by the accompanying status byte. This will determine the type of message that is sent (as they are normally channel messages) between note on, note off, pitch bend, or another type of message. Within a status byte, the MSB begins the MIDI message, then the next three bits actively determine the type of MIDI message that is being sent. There are eight possible combinations which determine the type of MIDI message:

$$000, 001, 010, 100, 101, 110, 111$$

Finally, the last four bits of a MIDI message determine which channel the MIDI message is delivered through. There are 16 possibilities, for each of the 16 channels

Channel number (0-15)	Note number (0-127)	Attack velocity (0-127)
(1001 CCCC)	(0NNN NNNN)	(0VVV VVVV)

Table 2.1: The structure of a note on and note off MIDI message

available in MIDI. It is through this combination of bits and bytes which creates each of the six major types of MIDI messages.

2.3.3.1 Note On Messages

The note on message indicates the beginning of a MIDI note. Typically, one note on message is generated each time a note is triggered on a MIDI device, such as a keyboard, controller, or other MIDI instrument[4]. A note on message will contain three bytes of information: the MIDI channel number, the pitch number, and the attack velocity value. As in Table 2.1, the first byte specifies that this MIDI message is a note on message, and the MIDI channel that this message will go through. The second byte determines the specific note, of the possible 128 notes numbered 0-127, which will be sounded by the MIDI instrument. The third and final byte of a note on message is the note's velocity, also ranged from 0-127. This will denote the loudness of the sounding note, increasing in volume the higher this value goes. For MIDI instruments which do not interpret the entire 128-numbered range of velocity values, we will instead see an attack velocity value of 64 used, regardless of how loud or soft the note itself may be played. This value of 64 gives the note a dynamic/loudness of *mezzo forte*, or moderately loud. Additionally, a note on message with an attack velocity of 0 will generally be equivalent to a note off message, discussed in the next section. In a note on message with an attack velocity value of 0, the MIDI receiver will generally silence the currently sounding note, by playing it with a velocity (or volume) value of 0 [18].

[4]This includes, pressing down a key, hitting a MIDI drum pad, or playing a MIDI sequence.

2.3.3.2 Note Off Messages

A note off message is similar to that of a note on message, except that it is a command used to stop playing a MIDI note. A MIDI note on message will play until a corresponding note off message for that note is received by the MIDI receiver. So, a musical composition can, in its most basic form, be written as various MIDI note on and note off messages. There are also three bytes in a MIDI note off message (refer to Table 2.1), except that the third byte instead is a release velocity value. This value, also ranging from 0-127, will indicate the velocity in which a key or controller is released. A lower value indicates that the key was released slowly, and a higher value shows the key was released quickly. For MIDI devices which are able to respond to receiving a release velocity value, these are able to be programmed to vary the note's speed of decay, reducing the note's decay time as the release velocity value increases [21]. Decay is defined as the overall level of a sound dropping from the high point after the Attack stage of an ADSR envelope. Typically, this refers to what happens after striking a drum or plucking the string of a guitar or violin, in which the peak volume of this strike decays in loudness from its initial level. Eventually, this level will decay down to silence [29].

2.3.3.3 Pressure/Aftertouch Messages

Channel number (0-15)	Note number (0-127)	Release velocity (0-127)
(1000 CCCC)	(0NNN NNNN)	(0VVV VVVV)

Table 2.2: The structure of an aftertouch MIDI message

Pressure messages, also known as "Aftertouch" messages, happen after a key is pressed, and the user decides to press down on the key harder. For compatible MIDI devices, aftertouch can generally be assigned to parameters which include vibrato (the pulsating or vibrating element of some sounds, produced by a very slight fluctuation of the pitch of a note), volume (loudness), filter cutoff (the frequency range which will not pass through the filter), and pitch (frequency). As defined by MIDI, and the byte structure of an aftertouch message in Table 2.2, there are two types of aftertouch messages: channel pressure messages and polyphonic key pressure messages [18].

Channel number (0-15)	Note number (0-127)	Pressure value (0-127)
(1101 CCCC)	(0NNN NNNN)	(0VVV VVVV)

Table 2.3: The structure of a channel pressure MIDI message

Channel pressure messages are messages commonly transmitted by instruments which only respond to a singular overall pressure, regardless of the total number of keys played. These messages also contain three bytes of information: the MIDI channel number, the note number, and the pressure value, as in Table 2.3 [18].

Polyphonic key pressure messages are similar to channel pressure messages, but respond to the pressure changes that are applied to the individual keys of a MIDI keyboard. A MIDI device compatible with this type of MIDI message is able to respond or transmit the individual key pressure messages of each key that is pressed down. A MIDI device may not always be compatible with polyphonic key pressure messages, so availability of this message type will vary. However, typically, a MIDI device will contain bindings for other performance parameters which include vibrato, volume, timbre (the quality of a sound, which is the component which causes different instruments to sound differently from each other while playing the same note [6]), and pitch [28].

2.3.3.4 Program Change Messages

Program Change messages are used to change the active program or preset number of a MIDI device. This preset number is a user- or factory-defined number to select

Channel number (0-15)	Program ID number (0-127)
(1100 CCCC)	(0PPP PPPP)

Table 2.4: The structure of a program change message

a specific sound patch or system setup, to alter the output sound. With this message, up to 128 presets (in accordance to the established 0-127 numbered range available in MIDI) can be selected as a preset [18]. Commonly used to switch between presets on a digitally controlled mixing console, change loaded sounds, and more, this message consists of two bytes of information: the MIDI channel number, and the program ID number (0-127), like in Table 2.4.

2.3.3.5 Pitch Bend Messages

The pitch wheel is a common module found on most electronic keyboards and MIDI keyboards. The sensitivity of a pitch bend message refers to the responsiveness level (in semitones) of a pitch bend wheel or other pitch bend controller. This message is encoded in two bytes [28], and yields a total of 16,384 distinct semitone steps to pitch bend. Thus, the range of this message extends from its bottom end of -8192 to +8192, with 0 at the center serving as the instrument's true unaltered pitch [18]. This gives the physical pitch bend wheel values between 0 and 127, with 64 as the middle true pitch, as possible value ranges.

2.3.3.6 Control Change Messages

The control change type of MIDI message is typically referred to as a performance controller, as it is capable of communicating with the many knobs and sliders on MIDI controllers. These messages relate the real-time control over these performance parameters. There are three main types of control change messages:

1. continuous controllers: controllers which relay a full range (0-127) of variable control settings.

2. switch controllers: controllers which have an "off" and "on" state, with no intermediate settings.

3. channel mode message controllers: controllers which range from the controller numbers 120 and 127, used to set the note's sounding status, the instrument reset, the local control's on/off, all notes off message, and the MIDI mode status of a device.

Channel number (0-15)	Controller ID number (0-127)	Corresponding controller value (0-127)
(1011 CCCC)	(0CCC CCCC)	(0VVV VVVV)

Table 2.5: The structure of a control change message

A control change message will be transmitted whenever the corresponding controllers are varied in real time [18], and consist of three bytes: the MIDI channel number, controller ID number (0-127), and the corresponding controller value (0-127), as in Table 2.5. The second byte of a control change message, which dictates the controller ID number, is used to specify which of the device's program or performance parameters are to be referenced.

The third byte of the control change message, the controller value, describes the controller itself's actual data value. This is used to specify the position, depth, and/or level of a particular parameter [18].

2.4 SuperCollider: A Domain-Specific Language

SuperCollider is a domain-specific programming language (DSL) and environment created in 1996 by James McCartney [25]. A domain-specific language is a programming language with a higher level of abstraction, and optimized to solve a specific class of problems. It will use the concepts from the specific field of expertise it is built and designed for. A domain-specific language will differ from a "normal" programming language in that it is usually less complex than a general-purpose language such as Java. Most often, these are built to be used by non-developers who are familiar with, and experts in, the domain that the DSL addresses.[5] SuperCollider was built for real-time audio synthesis and algorithmic composition.[6] Since 1996, SuperCollider has evolved into an environment that is actively used and developed upon by both scientists and artists working with sound. In 2002, it was released as open-source software under the GNU General Public License.

The SuperCollider environment itself is made up of two parts: the server (*scsynth*) and the client (*sclang*) [25]. The SyuperCollider server, scsynth, is a real-time audio server which forms the core of the SuperCollider platform [27]. The client, sclang, itself is unable to perform sound synthesis, so it will send the commands for synthesis to the scsynth server, where scsynth performs the sound synthesis and audio output.

The server *scsynth* renders audio, and works similarly to an analog modular synthesizer, where the audio output of a chain of various modules can be routed into another module. Audio in scsynth is created through graphs called Synth Definitions (SynthDefs) [25], which are definitions of synths (synths create a singular sound producing unit), but also able to do anything audio-related within Super-Collider. Before exploring SynthDefs, we first must understand Unit Generators. In SuperCollider, Unit Generators (UGens), as in Listing 2.1, are the key building blocks for digital synthesis. Just like black boxes in coding, UGens contain complex calculations, and move them into a simple black box which outputs the synth builders. These are modular, and so the output of one UGen can serve as the input for another.

[5] More information can be found in JetBrains' article about DSLs `https://www.jetbrains.com/mps/concepts/domain-specific-languages`

[6] This project does not focus on algorithmic composition, which put most simply is a method of composing music using an algorithm or multiple algorithms.

```
// A sine wave Unit Generator, playing at a frequency of
    ↪ 440 Hz
{SinOsc.ar(440, 0, 1)}.play;
```

Listing 2.1: A simple sine wave unit generator

A SynthDef is a pre-compiled graph of UGens. To turn a Unit Generator into a SynthDef, we start with a simple synth, and place it into a UGen. As in Listing 2.2, we begin with a simple synth, this time a sawtooth wave. Then, we give a name, mysaw, to the function which will call the UGen. The sawtooth wave is wrapped in a UGen called "Out." We then create a Synth, or a child (instantiation) of a SynthDef, which can be controlled by referencing it with a variable, as in Listing 2.3 [27].

```
// This is a simple synth, a sawtooth wave
{Saw.ar(440)}.play

// This will become this synth definition
SynthDef(\mysaw, {
 Out.ar(0, Saw.ar(440)));
}).add;
```

Listing 2.2: A simple synth, wrapped into a UGen

```
// We create a synth using the \mysaw function, and place it
    ↪ into variable 'a'
a = Synth(\mysaw);

// We create a second synth with \mysaw, and place it into
    ↪ variable 'b'
b = Synth(\mysaw);

```

```
a.free;
b.free; // frees both synths a and b, to free the resources
    ↪ associated with them before the program closes
```

Listing 2.3: Instantiating a SynthDef to create a Synth

The client, *sclang*, is an interpreted programming language. It controls scsynth using Open Sound Control (OSC).[7] It is a dynamically typed, garbage-collected, single-inheritance object-oriented and functional language [27]. With syntax similar to the C programming language, for a developer, the architecture of the language strikes the balance between the needs of realtime computations and the flexibility and simplicity of an abstract language. With the similarity between sclang and languages such as Lisp and the C language family, the semicolon and brackets are important to running a SuperCollider program. Brackets in SuperCollider help create a scope within the program for the interpreter. So, as in Listing 2.4, the following will not run all at once, unless each individual line is highlighted then set to run.

```
var freq = 440;
var amp = 0.5;

{SinOsc.ar(freq, 0, amp)}.play;
```

Listing 2.4: A basic example of SuperCollider code

Brackets make a program's compilation and runtime much easier to handle. When we run code that is inside brackets, the code will be run all at once, as in Listing 2.5.

```
(
 var freq = 440;
```

[7]Like MIDI, OSC is a communication protocol used for audio.

```
  var amp = 0.5;
  {SinOsc.ar(freq, 0, amp)}.play;
)
```

Listing 2.5: Using brackets in SuperCollider

CHAPTER 3

Physics of Sound Waves

3.1 Mathematical Background

According to Joseph Fourier, the creator of the Fourier Transform, any periodic signal or sound can be reduced into their individual sine waves, or other waveform types [5]. There are five basic periodic waveform types: a sine wave, square wave, sawtooth wave, triangle wave, and pulse wave [47]. Each of these waveforms, except for the pulse wave, are sinusoidal waveforms which repeat in a pattern of motion known as a cycle. From this, we calculate the *period* to be the time length between a cycle. The pulse wave is a special type of waveform, with its own individual characteristics, different from the other four periodic waveform types. As a non-sinusoidal waveform, or a wave which does not oscillate similar to the repetitive oscillation of a sine wave, it may assist in the creation of square waves, a waveform which is periodic and sinusoidal in nature. However, for the purposes of this work, the pulse wave will not be discussed within the scope of this project, due to its nature as a non-periodic and non-sinusoidal waveform, two waveform types which are not included in this synthesizer. The remaining four periodic waveforms each also have an individual sound and characteristics, and thus will be used for different types of sound synthesis and applications.

In its most basic form, there are three general types of waves [16].

1. Mechanical waves.

2. Electromagnetic waves.

3. Matter waves.

Mechanical waves are the most familiar to many, due to its ubiquity; mechanical waves are found in water waves, sound waves, and sonic waves. These waves are identified by two key features: a governance by Newton's three laws, and an existence that occurs only within a material medium, such as air or water. Additionally, these waves follow the idea of *simple harmonic motion*, in which the wave oscillates in a specific path, and varies sinusoidally in time, following a sine or cosine function.

Electromagnetic waves (also defined as "light" or "light rays") are the second most familiar type of wave, found in visible and ultraviolet light, x-rays, and radar waves. Unlike mechanical waves, electromagnetic waves do not require any material medium to exist, able to travel through both the vacuum of space, and mediums such as air or glass. Albert Einstein (1879-1955), a German theoretical-physicist, introduced the *Theory of relativity* sometime between 1905-1915, which discussed the theories of the structure of spacetime and of gravitation [16], and would later include the special characteristics of electromagnetic waves. As light was discovered to contain the same speed regardless of the frame of reference from which it was measured, its speed was measured to be the exact value of $c = 299,792,458\ m/s$ in a vacuum [16].

Finally, matter waves are the least frequently recognized type of wave. These waves work with the fundamental types of particles, including protons, electrons, atoms, and molecules.

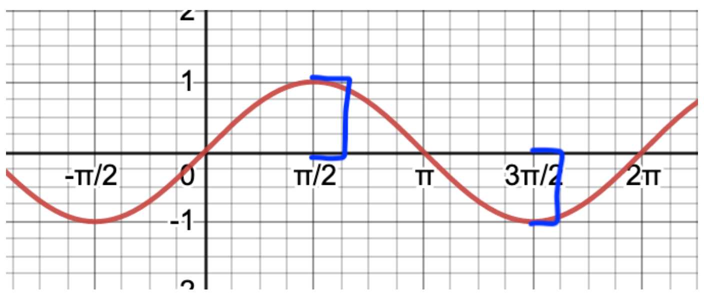

Figure 3.1: A sine wave, with the peaks and troughs of the amplitude marked blue

The speed with which each wave rises and falls is its frequency f. If the frequency is too low (less than 20 cycles per second, or 20 Hertz, abbreviated as Hz), little to no noise will be audible by the human ear. If the frequency is too high (generally above 20,000 cycles per second, or 20 kilohertz, abbreviated kHz), again, few noises besides high-pitched and shrill noises will be audible. The range of human hearing is generally stated as being from 20 cycles per second, with 20 Hertz at the low end, to 20,000 cycles per second (20 kHz) at the high end. As people age, they generally lose the ability to hear the higher frequencies.

For any sine wave, like one seen in Figure 3.1, one method of notating the wave function is in Equation 3.1 [16], describing a sine wave which oscillates parallel to the y-axis at time `t`, with the displacement of y (the level of the wave's peaks and troughs, marked in blue of Figure 3.1) at position x, variables which will be consistent across all other waves presented, and seen in Appendix B.

$$y(x, t) = y_m \sin(k(x + \lambda) - \omega t) \tag{3.1}$$

The amplitude of a wave (y_m) is the magnitude of the maximum displacement of the wave's crest (peak, above 0, or trough, below 0) from its equilibrium position of 0 on the x-axis. As this value is a magnitude, the quantity of amplitude will always be positive, even if it measures the trough of a wave (a negative value, with the amplitude of the wave below 0) instead of the peak (a positive value, above 0). The phase of the wave (defined as the position of a point in time t in a waveform cycle, and in Equation 3.1 notated as $kx - \omega t$) will change linearly with a time `t`, dependent on the oscillation of the wave. Arbitrarily, as a sine wave oscillates between values of -1 and $+1$, the wave's amplitude is at value $-y_m$ and $+y_m$ respectively. Thus, the time-dependent nature of the wave's phase will correspond to the oscillation of the wave, with the amplitude determining the extremeness of the displacement of the wave's crest. The wavelength λ of a wave is the distance between repetitions of peaks or troughs, and is parallel to the direction of the wave's travel. Finally, the period of a wave `T` is the time to move through one full oscillation [16]. Due to the nature of a sine wave to follow the unit circle counterclockwise (discussed further in Subsection 3.1.1), the sine wave will begin to repeat when its angle θ (or argument k) is increased by 2π. Thus, we have that Equation 3.7 is equivalent to Equation 3.2.

$$k = \frac{2\pi}{\lambda} \tag{3.2}$$

It is important to note that there are two types of frequency within sound waves: ω (angular frequency) and f (temporal frequency). The temporal frequency f of a wave is defined relative to angular frequency ω, as in Equation 3.3. Frequency f is the number of oscillations per unit time, usually measured in Hertz or kilohertz. Angular frequency ω is the frequency of an arbitrary sine wave as it moves counterclockwise around the unit circle.

$$f = \frac{1}{T} = \frac{\omega}{2\pi} \tag{3.3}$$

The final notable property of waves is the wave forms (the shape of the waves, and different from the term "waveform") as the waves move left to right. Alternatively, we could also monitor the wave as it oscillates up and down, but the result is the same: either the displacement of the peaks and troughs of an oscillating wave is perpendicular to the direction of travel of the wave (defined as a *transverse wave*), or the direction of travel of an oscillating wave is parallel to the displacement of the peak of the wave (defined as a *longitudinal wave*) [16], as in Figure 3.2. Both wave shapes are also known as *traveling waves*, as they travel from one defined point to another. The sinusoidal, periodic waveforms mentioned in Section 3.1 (and further discussed in Subsections 3.1.1 through 3.1.4) are transverse waves; as in Figure 3.2, the black vertical arrows demonstrate the displacement of the sinusoidal wave along the y-axis, while the pink arrow (labeled $\vec{v}$) shows the movement of the wave along the x-axis.

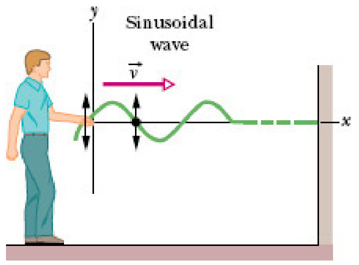

Figure 3.2: A sinusoidal wave is sent along the string. A typical string element moves up and down continuously as the wave passes. This too is a transverse wave.
[16]

Simple tones, such as sine waves and cosine waves, are transverse periodic waves–waves in which the displacement of the wave is perpendicular to the direction of travel of the wave, and which also repeats in a continuous pattern, known as a cycle–that are singular in frequency. These waveforms, as mentioned, are the simple waveforms which compose the building blocks of more complex sounds (waveforms). To demonstrate this concept, we choose the sine wave to serve as an example of the principle of *superposition*, defined as the situation in which two or more waves pass simultaneously through the same region [16], and several effects (sound alterations) may occur simultaneously, as the net effect is the sum of each wave's individual effects. Suppose we have two waves which are traveling simultaneously through the same stretched string, and let $y_1(x,t)$ and $y_2(x,t)$ be the displacements the string will experience if each wave were to travel alone the wave alone. The total displacement of the string when the waves overlap will then be

$$y'(x,t) = y_1(x,t) + y_2(x,t) \tag{3.4}$$

Occasionally, multiple sinusoidal waves of the same wavelength and amplitude moving in the same direction may *interfere* with each other, as the superposition principle applies. The resultant wave of this interference depends on the extent to which the waves are in phase/in step with respect to each other. If the phase of both waves are the same ("in phase," in which the peaks and troughs of each wave are the same) then the total displacement is doubled from the displacement of each individual wave, resulting in a sound that sounds twice as loud as either of the waves would individually. On the other hand, if both waves are exactly "out of phase" (the peaks of one wave is the same as the troughs of the other) then the total interference will result in the cancellation of noise, as each wave will fully cancel the other [16]. This is how much of the "noise-cancelling" technology works in headphones and speakers, in which the phases of multiple waveforms combine to cancel each other, resulting in the perception of no noise. As in Equation 3.5, suppose $y_1(x,t)$ is the displacement of y of one wave, and $y_2(x,t)$ be the displacement of another wave.

$$\begin{aligned} y_1(x,t) &= y_m \sin(kx - \omega t) \\ y_2(x,t) &= y_m \sin(kx - \omega t + \varphi) \end{aligned} \tag{3.5}$$

Waves $y_1(x,t)$ and $y_2(x,t)$ have the same angular frequency ω (and thus also the same temporal frequency f, the same angular wave number k (and thus also the same wavelength λ), and amplitude y_m. Both waves travel in the positive direction of the x-axis, at the same speed. The only difference between the two waves is the phase shift φ, resulting in the two waves being out of phase, and cancelling the noise of the resultant wave. In situations other than perfect “in phase” or “out of phase” waves, the composite sound will be some sum or difference of the amplitude, wavelength and phase of each of the individual waves.

3.1.1 Sine Waves

The first of the basic periodic waves is the sine wave, and is the most common type of periodic wave. The sine wave is a signal with only one frequency, and represents the unidimensional motion for any signal with a phase angle that rotates at a constant rate, using both the unit circle in Figure 3.4, and the trigonometric sine function of Equation 3.6. On the unit circle, the trigonometric sine function of a phase angle θ is defined as the ratio of the length of the opposite side and the hypotenuse of a right triangle, like in Figure 3.4 in which the right triangle is outlined in red. The unit circle, with a radius of 1, results in the sine function $sin\theta$ being equal to the y-value in Cartesian coordinates, where the hypotenuse of the right triangle that is formed meets the circle, like in Figure 3.3 and Figure 3.4. Here, when $y = 1$ for angle $\theta = \frac{\pi}{2}$ in the unit circle (Figure 3.4), the y-value of the wave in Figure 3.3 corresponds to this y-value, at 1. We can then use this trigonometric sine wave to synthesize a sine wave audio signal. As a sine wave is a continuous periodic wave, in which the wave continues to move until stopped, we must use the sine wave function on the unit circle continuously. Thus, we use the sine function continuously around the unit circle, going counterclockwise. We notice that in the correlation between Figures 3.3 and 3.4 moving counterclockwise through the unit circle results in the appropriate rise and fall of the sine wave. $\frac{\pi}{2}$ is the highest y-value within the Cartesian plane, and so denotes a peak in the sine wave, while $\frac{3\pi}{2}$ is the lowest, denoting a trough.

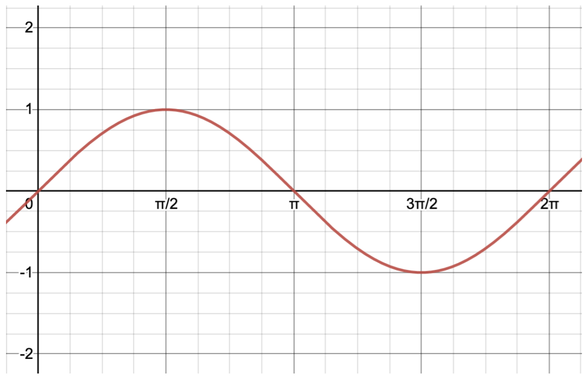

Figure 3.3: A basic sine wave

$$y = Asin(B(x + C)) + D \tag{3.6}$$

Like with the other periodic waves, sine waves have three important properties: frequency, amplitude, and phase. From the generic function for a sine wave, as in Equation 3.6, we are able to compute the various properties. First, A is the sine wave's amplitude. This is defined as the height from the x-axis of the wave to its peak or trough. For our unit sine wave, this value will be 1. Second is the variable B, which helps to define the period of the wave, or the distance between one peak and the next, or one trough and the next. With Equation 3.7, we see the period is equivalent to taking the total circumference of the unit circle, and dividing it by B. Third, the phase shift of a sine wave is denoted by C. If the expression is $(x+C)$, then the phase of the sine wave will shift to the left with respect to the value $x + 0$ (with $C = 0$), as the x-value of the wave becomes positive C. Otherwise, if the expression is $(x - C)$, then the wave will shift right with respect to the value $x - 0$ ($C = 0$), as x becomes negative C. Finally, the variable D is equivalent to the vertical shift of the wave. This notates the distance that the wave will shift vertically from its unit circle position. As sine waves are only composed of a fundamental frequency (the lowest frequency found in a wave, for sine waves this will be the frequency of the wave), these waves are most used as pure sound tones. Sine waves are at the center

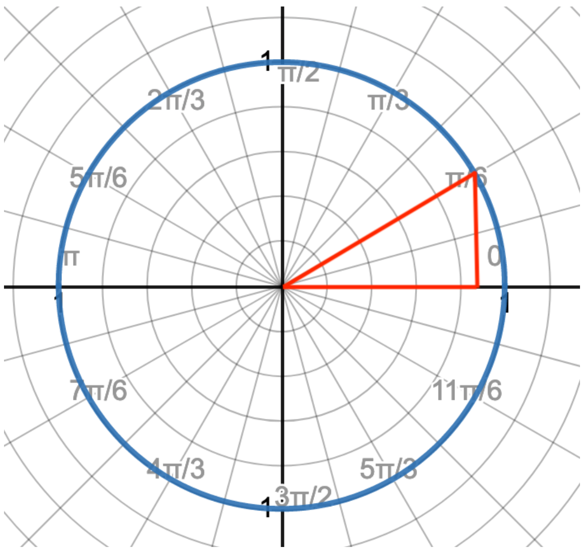

Figure 3.4: The unit circle

of audio synthesis and sound analysis. Other, more complex, periodic waves can be built using some combination of sine waves.

$$\frac{2\pi}{B} \tag{3.7}$$

3.1.2 Square Waves

The square wave is a certain type of pulse wave, which we can represent as a summation of an infinite set of sine waves, similar to how sawtooth waves and triangle waves can also be a summation of sine waves. *Additive sythesis*[1] is one method we can use to create square waves, sawtooth waves, and triangle waves, and so we notice that the functions for these waves have similar periodic equations. Through additive synthesis, we layer multiple sine waves, such that the composite wave is typically a square wave, sawtooth wave, or triangle wave. In the equation for a square wave, we have variables f (frequency), A (amplitude), and n number of harmonics, let $f(x)$ be an arbitrary square wave as a function of time t in Equation

[1]Further defined in section 2.1.

3.8. Like with triangle waves, odd harmonics (or waves in which only odd values are valid) are the only valid values for this type of periodic waveform. This causes a non-smooth waveform, as unlike the smooth oscillations of a sine wave, odd waveforms ramp upwards to the peak amplitude height, and downwards to the lowest amplitude trough. This results in a sharper sound. Then, positive i, in Equation 3.8, will always be odd, such that these harmonics create the square wave of Figure 3.5.

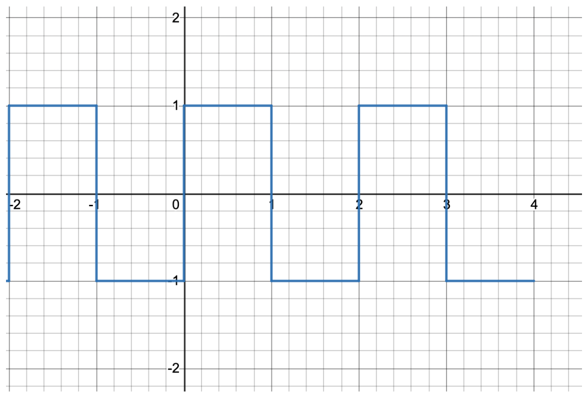

Figure 3.5: A basic square wave

$$f(x) = A \sum_{i=1}^{\infty} \frac{\sin(2\pi n f t)}{n} \tag{3.8}$$

There are three properties of the square wave which we can manipulate: the frequency f, amplitude A, and phase (altering $2\pi n f t$ from Equation 3.8), from the summation equation for square waves (Equation 3.8). First, the amplitude value A (similar to variable A of Equation 3.6) will alter the height (displacement of y) of the wave's peak and trough from the x-axis. This will change the volume we perceive the square wave to be at. In Figure 3.5, the square wave pictured is a unit square wave, in which the amplitude will be 1. Second, is frequency f, which defines the period of the wave. A larger f (like with variable B of Equation 3.6) value will cause the value of the period also becoming larger, thus resulting in a sound with a higher pitch. Third, the phase shift of the square wave will be denoted by changes to the expression $2\pi nft$ (similar to the usage of variable C of Equation 3.6). Like with changes to C of the sine wave equation, alterations to a square wave's phase involves adding or subtracting a value to time `t` [41]. For instance, to adjust the phase of a square wave to sound 0.5 seconds delayed, we would alter $2\pi nft$ by doing the following

$$f(x) = A\sum_{i=1}^{\infty} \frac{sin(2\pi nft - 0.5)}{n}$$

As the value of this modification to time `t` is negative (-0.5), the wave will shift right, and sound delayed. This is similar to the negative C value of the sine wave function, since a positive value will result in the square wave to shift to the left (in respect to a phase shift value of 0) and sound early.

The sound produced by a square wave is one that is harsh and bright, that is, a sound which has many frequencies in the higher frequency ranges (bright), and lacks much of the lower end of the frequency ranges (harsh). In combination, this results in a sound that becomes grating to hear after some time, and introduces listening fatigue. These waves also only contain odd harmonics, so are most used to simulate the sounds of woodwinds and reeds [10].

3.1.3 Sawtooth Waves

Like with the square wave, one notable characteristic of the sawtooth waveform is that is it a non-sinusoidal wave, and resembles the teeth of a plain-toothed saw. Additionally, as in Figure 3.6, a sawtooth wave will ramp upwards to a peak amplitude height as the amplitude increases linearly, reaches a set maximum volume, then sharply drop to its trough, or the set minimum value (typically 0). This sudden drop (the non-linear component of sawtooth waves) to the set minimum value for amplitude results in the wave beginning a new period, or new cycle. This is defined by Equation 3.9, in which multiple sine waves create a sawtooth wave [44]. We notice that this equation is similar to that of the square wave function, with a small difference: the expression $2\pi fnt$ is no longer a fraction over n to create a sawtooth wave. This difference is seen in both Figure 3.5 and Figure 3.6, in the ways in which each wave descends from its peak to its trough, and the value of time `t` which the wave spends at its peak or trough.

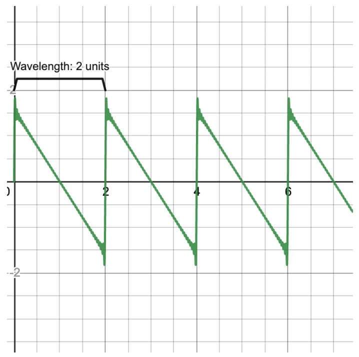

Figure 3.6: A basic sawtooth wave

$$f(x) = \sum_{n=1}^{\infty} \frac{A}{n} \sin(2\pi fnt) \tag{3.9}$$

In a sawtooth wave, we again have three properties we can easily manipulate: amplitude (A), frequency f, and phase shift (the expression $2\pi fnt$). Each of these variables have their sine wave function equivalent from Equation 3.6. Amplitude A equals the sine wave function variable A, frequency f is equivalent to variable B, and phase shift $2\pi fnt$ is the same as variable C. The amplitude A will alter the sawtooth wave's amplitude, adjusting the distance from a peak or trough to the x-axis, and thus also the perceived volume. Frequency f defines the period of the sawtooth wave, in which a higher value f will cause a higher number of cycles to occur, and the perceived pitch to rise. Finally, the phase shift $2\pi fnt$ works similarly to how it does in the square wave function, in which an addition or subtraction to time t affects how early or delayed the resulting sound will be.

Within physical musical instruments, sawtooth waves are most often found within the playing of a string instrument, such as the violin, viola, or cello. As a bowed string oscillates, the bow alternates, in an up-down motion, sticking to the string. The bow will drag the string along as it plays, then slips to allow the string to return to its neutral position [20] Thus, the nature of the sawtooth wave gives it a bright and energetic sound, and contains both even and odd harmonics, creating a relatively smooth oscillating sound. Due to this, sawtooth waves will be used for a variety of instruments, such as strings, brass, pads, and basses [10].

3.1.4 Triangle Wave

A triangle wave is also a non-sinusoidal waveform, named for its triangular shape. Like with square waves, this type of wave only contains odd harmonics, creating a sound which is not smooth in its oscillation from peak to trough and back again. It also shares similarities to the sawtooth wave with the amplitude of the wave increasing linearly to a set maximum value from 0, reaching this maximum value, then decreasing back to 0. However, there is a small difference in the two waves. While a sawtooth wave will decrease in amplitude from its set maximum value back to 0 immediately (in a non-linear way, as seen in Figure 3.6) the amplitude of a triangle wave will decrease linearly upon achieving the set maximum amplitude value, as in Figure 3.7, until the minimum value is reached [44].

As with the other waveforms mentioned, triangle waves can be written as the summation of sine waves. Let $f(x)$ be any triangle wave as a function of time t in seconds, with frequency f, n number of harmonics (which can simply be thought of as cycles) and amplitude A. Thus, Equation 3.10 will indicate the cycles of a triangle wave. The equation for the triangle wave is similar to those for the other

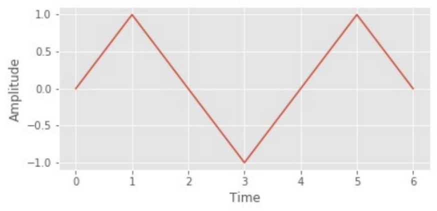

Figure 3.7: A basic triangle wave
[41]

two waveforms (square waves and sawtooth waves), with the primary differences being the multiplication of $(-1)^i$ and the amplitude A being divided by $\frac{n}{2}$. So trough adjustments to variables f, A and $2\pi fnt$, various sound output changes can be achieved. First, the amplitude A will be divided by $\frac{n}{2}$ (the square of the number of harmonics n), and is similar to the variable A of Equation 3.6. This value determines the amplitude of the triangle wave, and thus also the perceived volume of the wave. Second, frequency f defines the period of the wave, similarly to variable B of Equation 3.6, affecting the number of cycles which appear and also the perceived pitch of the wave. Finally, the phase shift of $2\pi fnt$ acts similarly as it does in the square wave and sawtooth wave functions. A value added to $2\pi fnt$ results in the wave sounding early, while a subtraction from $2\pi fnt$ causes the wave to sound delayed.

$$f(x) = \sum_{i=1}^{\infty} (-1)^i \frac{A}{n^2} \sin(2\pi fnt) \tag{3.10}$$

Triangle waves are also "soft" sounding waveforms, which decrease in sound faster than some other types of waves. This results in triangle waves being used to simulate piano and flute sounds, as well as other instruments which rely on the quick decay of a note [10].

3.2 Representing Sound Digitally

We now understand the physics and mathematics behind sound waves. However, sounds in real life are not composed of simple pure waveforms; sounds in the world around us are made up of multiple harmonics and frequencies layered on top of one another to produce the composite sound which we hear. Sound, like electricity and light, is a form of energy in which molecules in air vibrate and move in a wave pattern. This wave pattern produces the sound waves [2]. Air is able to support multiple sound waves simultaneously, explaining our ability to hear different sounds at the same time. This sound energy is dispersed outwards from the sound source, and will continue to move until the molecules run out of energy, as the energy weakens the further it moves away from the sound source (or the sound *attenuates*). The sound energy is transferred between molecules, as each molecule moves from an original resting point, transfers energy to another molecule as they collide, then returns to its resting point, as in Figure 3.8. This molecule movement is the oscillation of sound. Molecules become closer together when vibrating, and crowd together in certain places, and thus there are fewer molecules in other places. This visualization of crowds of molecules can be done as a wave, with the peak of a sound wave indicating that there are more molecules together in space (compression of air molecules), and the trough of a sound wave indicating there are fewer molecules (rarefaction) [45]. Typically, sound is visualized in a graphical format, with peaks and troughs to a wave rather than drawings showing the compression and rarefaction of air molecules.

Through this periodic nature of waves, the repetitions create what we will recognize to be musical sound, but lacking the specific tonality and timbral qualities of specific instruments. Musical instruments generate a composite set of frequencies, arranged as a layered set of harmonics above the fundamental frequency (the lowest frequency which is played). Thus, the fundamental frequency will be the pitch of

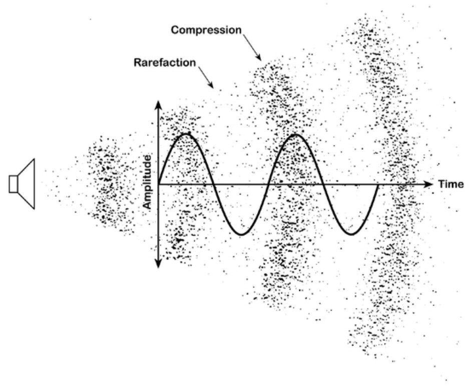

Figure 3.8: A graphical representation of sound, vs its physical phenomenon [45]

the note, and the additional harmonics (also known as overtones) will lie above this pitch and add the tonality or timbre of the note [45]. This is what differentiates pure sound waveforms from notes generated from musical instruments, as pure sound waveforms all have pitch, but lack specific timbral quality.[2] Another factor which dictates what a sound wave will be perceived as is the amplitude of a wave, which will be further discussed in Section 3.3.

To get from an analog sound wave to a digital sound signal, we must sample the continuous analog wave into its digital representation. Current technology is incapable of working with a complete, continuous analog sound wave, as it is made up of infinite points. To counter this, we must sample the analog sound wave to convert it into a digital signal, which produces a finite set of discrete points. Together, these points will allow us to visualize an analog sound wave in the digital domain. To sample an analog audio signal is to measure the signal at different points in time, while recording the physical property of that signal as a number. Typically, voltage is measured, which can be transmitted into a stream of numbers [1]. While an audio signal or sound is a continuous set of values, which are able to be displayed on an oscilloscope (a tool to show oscillations) as a waveform, the

[2]This is what gives pure sound waveforms their name, "pure tones," as each of these waveforms has pitch but no timbre.

digital “signal” is a series of numbers, representing various discrete values from the continuous wave. These numbers will represent the values of an audio signal at specific points in time, and are known as *samples* [39]. The sampling process to convert from analog (physical) waves to a digital signal involves two steps:

1. The audio signal is “sampled” in which data is captured at defined intervals from one continuous audio signal.

2. The sample values are converted into discrete numbers.

Once the sampling stage is complete, this sampled data will be digitally scaled down to a target data rate and bandwidth for further processing and storage. The process of sampling will typically produce numbers which are an incomplete representation of the original audio sound, but through careful sampling, this amount of incompleteness can be made insignificant. This same process can be reversed to convert from a digital signal to an analog one, and is known as “sample replay.” Sample replay also has three stages and serves as the basis for digital synthesizers, as the conversion from digital signal to analog wave produces the sound that is heard.

Analog sound exists in two dimensions: time (a period of each wavelength) and space (the displacement of each waveform from the atmospheric pressure). So it makes sense that to convert from analog audio to digital signals, we use these two dimensions: time (to convert a continuously moving analog audio wave to the digital domain, through the use of a *sample rate*) and space (in the digital domain, this will be in terms of the *bit depth*).

Sample rate, or *sample frequency*, is the number of samples which are captured per period (which is commonly measured in Hertz). Typically, digital audio samples of analog audio waves are taken at 8,000 times per second, or more [48]. Occasionally, a sound will be sampled that is at a frequency rate higher than 8,000 and the system doing the sampling is unable to identify two points on the waveform to understand the period. Harry Nyquist (1879-1976), a Swedish physicist and electrical engineer, solved this problem through the *Nyquist frequency* [48] (or *folding frequency*). Nyquist discovered that in order for an analog audio wave to be accurately represented by a digital signal, and thus also the system which is sampling the audio wave, the wave must be sampled at least twice per wavelength, known as the *Nyquist Theorem* (or *Nyquist-Shannon* theorem) [48] and visualized in Figure 3.9. Within the first part of Figure 3.9, the signal is sampled perfectly at 2*x* the Nyquist Frequency. This allows the relevant discrete numbers to be captured, to fully represent the wave. The second image of Figure 3.9 is sampled at a rate which is less than 2*x* the Nyquist Frequency (1.5*x* for the middle image), managing to capture less than half the needed discrete values to fully represent the wave. Thus, for a given sample rate, the highest frequency within a system cannot exceed half this sample rate. In other words, the Nyquist frequency is the frequency at 50% of the sample rate. Otherwise, for a frequency which does not satisfy the requirements for the Nyquist Theorem, a phenomenon called "aliasing" would occur. Aliasing is the phenomenon in which signals that exceed half the sample rate are misrepresented as "alias signals" leading to the wave to be indistinguishable, and incorrect frequency and amplitude values to be obtained. The middle image of Figure 3.9 describes the concept of aliasing, in which the wave is sampled at only 1.5*x* the Nyquist Frequency, and alias discrete points are introduced. These alias points, which make up an "alias signal," are not a true representation of the original signal. An "anti-aliasing" filter, or Nyquist filter, would then typically be applied to the signal before sampling occurs, to act as a low-pass filter with a cut-off at the Nyquist frequency.

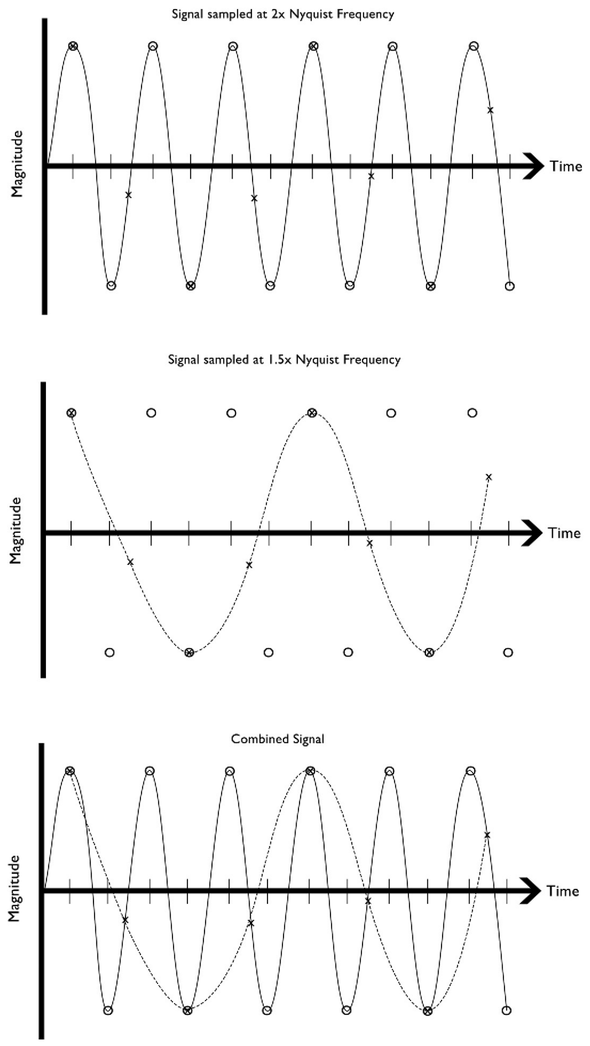

Figure 3.9: The Nyquist sampling theorem visualized. Upper: Signal sampled at 2x Nyquist frequency, Middle: Signal sampled at 1.5x Nyquist frequency (and thus aliasing and not a true representation of the original signal), Bottom: Signal containing both accurate and aliasing components

[48]

This sampling process using the Nyquist frequency and anti-aliasing filter is commonly used in digital audio systems (both for professional and consumer usage). Another tool that is commonly used to improve the anti-aliasing filter is *oversampling* [19]. Oversampling can increase the effective sample rate between 12 and 128 times the original rate, with three primary reasons for this process.

1. Physical anti-aliasing filters–like those found in analog-to-digital converters

(ADC)–can be expensive, and difficult to design properly. So by increasing the effective sample rate, and the ceiling for this sample rate, a simpler and less-costly filter can be used.

2. Oversampling typically results in a higher-quality conversion, for both analog-to-digital (A/D) and digital-to-analog (D/A) conversion.

3. Multiple samples are usually taken of a single sample rate, and so the average noise level will be lower.

Bit depth is defined as the discrete values which are available within the digital system, representing the magnitude of a continuous electrical signal [48], in the form $log_2 x$. These levels can become exponentially large, and so *quantization* is used to process data from a broad range into values within a smaller range. It will convert the levels of a continuous analog signal, as the signal after sampling is still in the analog domain, into binary digits (bits). Using bits will allow us to be able to manipulate and store audio data digitally. After sampling the amplitude of an analog audio wave at various precise intervals, we are able to output this amplitude level in its equivalent in bits, which represent the originally sampled amplitude level, known as *bit rates*. The bit "rate" can then be defined as the amount of data stored or transmitted per second of time, or in Equation 3.11, measured in kbps [48]. In general, the higher the value of the bit rate, the higher the quality of the audio data that is stored, since there is more data to represent the captured sound available.

$$\text{bit rate (bits per second)} = \text{sample rate (Hz) x bit depth x number of channels} \tag{3.11}$$

Increasing the number of channels will increase the *stereo width* of a sound, or how "wide" it is. *Stereophonic sound* (or *stereo*) is a method of sound reproduction which recreates a multi-directional, 3-dimensional audible perspective. Typically, this is achieved by using two independent audio channels fed through a configuration of two loudspeakers (or stereo headphones) to create the impression of sound heard from various directions, similar to how we naturally hear [2]. Different sounds will play from the left speaker or headphone than from the right, thus creating the illusion that the audio is coming from multiple directions. Stereo sound will be used most often in film and music, where there is a combination of music, vocals, and other sounds. *Monophonic sound* (*mono*) occurs when two loudspeakers, or stereo headphones, play the same sound equally [2]. Both the left and right speaker or headphone will play identical audio. Mono sound will typically be used for radio talk shows, telephone calls, or other times when there is a focus on vocals.

From our original three stages of converting analog sound to digital signals, we can now specify the processes which occur in each stage [19].

1. Sampling analog audio wave amplitude levels at precise intervals in time.
2. Converting these samples into the digital bit value (typically a 16-bit length), which most accurately represents the amplitude levels.
3. Storing these sample equivalents (the bit values) within digital memory.

Upon playback, the bits are then converted back into analog amplitude values, at precise intervals in time, allowing for the originally recorded amplitude values to be re-created, processed, and played back [19].

3.3 Manipulating Sound Waves

As previously mentioned in Section 2.2, modular synthesis involves sending an audio signal through patches or modules in a linear format to achieve the desired sound output changes. To explain how these changes occur to a sound wave, we arbitrarily choose the simple sine wave as an example, as in Appendix C.

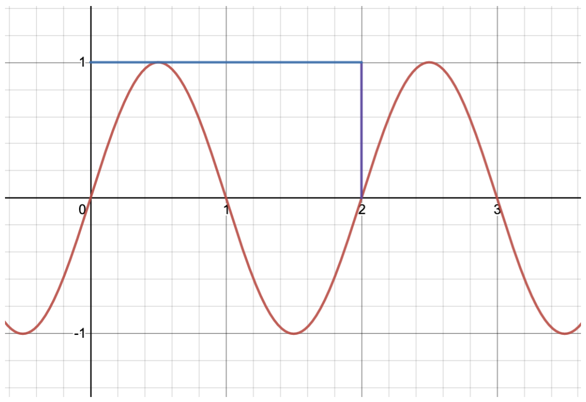

Figure 3.10: A basic sine wave, with period demarcated in blue, and amplitude in purple

$$y(t) = A\sin(2\pi f t + \varphi) \tag{3.12}$$

Sine waves are a waveform which is a function of time t with variables A, φ. A is the wave's amplitude, which determines the peak deviation from zero. The frequency is f, and φ is the wave's phase, which specifies (in radians) where in its cycle the wave's oscillation is at time t [21]. When φ is not equal to zero, the wave itself will appear to be shifted by the value equal to φ, which is known as a wave's "phase shift." A negative value will represent a delay in sound, while a positive value will represent an advance in the heard sound.

Thus, there are three primary parameters when manipulating audio (or a simple waveform); amplitude, frequency, and phase can all be modified at various points to affect the audio output. The first, amplitude, will determine the volume of the wave's sound. The larger the distance between zero and the wave's peak, the louder the human ear will perceive the sound to be [48]. In Figure 3.10, amplitude is colored purple, and we see it has a value of 1 (the default value of amplitude of a sine wave from the unit circle), as A does in Equation 3.12. By changing the value of A to either $\frac{1}{2}$, or 2, the peak of the wave will change accordingly, becoming larger or smaller depending on the set value of A. This change is reflected in the sound we can hear, as like in Figure 3.11 and Equation 3.13, the volume of this sine wave is halved. Thus, volume ranges can be between soft (at a barely audible *pianissimo*) with an amplitude $A = \frac{1}{2}$, or loud (*fortissimo*), with $A = 4$, for instance.

$$y(t) = \frac{1}{2}\sin(\omega t + \varphi) \tag{3.13}$$

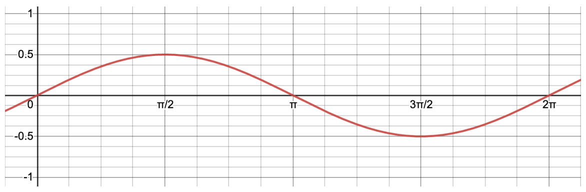

Figure 3.11: A sine wave, with an amplitude of $\frac{1}{2}$

The second option commonly used to manipulate audio is to change an audio signal or wave's frequency. For sound and audio manipulations, the frequency component determines a sound's "color," or "timbre." It is the property of a waveform which determines the output sound's pitch. (Refer to Appendix A for a review of the range of audible frequencies.) Thus, with Equation 3.12, we change the value of w, which increases the rate of change of the sine wave, increasing the perceived pitch. With the period of the sine wave in Figure 3.10 marked blue, it is this blue section that will increase with an increase in ω. As ω increases, there are more repetitions of the sine wave's phase, so the audio output's pitch will increase. Pitch is how high or low a sound is perceived to be, and will be determined by the frequency of the vibrations [45]. Frequency is the number of wave cycles which pass through a given point per second. A higher frequency will result in a higher pitch, and a lower frequency will result in a lower pitch. For instance, with a frequency of 261 Hz, we perceive the note Middle C to be played.

Finally, a modification to a wave's phase will determine if the audio signal output is on-time, delayed, or early. The numeric value of φ depends on the start of the wave's period. Similar to the changes made to amplitude and frequency, by modifying the value of φ, we change the phase of the waveform. This will be most noticeable with multiple harmonics or simple waveforms stacked on top of each other, in which each signal will have a phase at a slightly different time, as in Figure 3.12. The period of the blue sine wave has a length of $\frac{\pi}{2}$, but otherwise is a normal unit circle sine wave. The red sine wave is phase shifted, with an φ value of positive 2, which shifts the wave negative and to the left, causing the output audio to sound early in comparison to the blue wave.

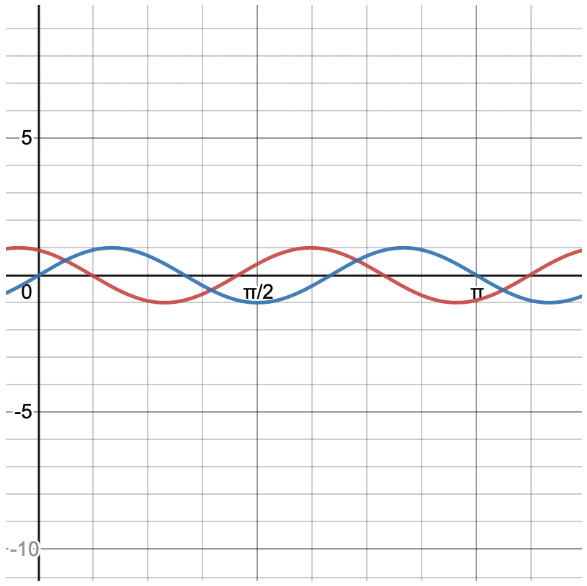

Figure 3.12: The phase shift in a sine wave

This type of sound modulation is done through a synthesizer's "delay" effect (sometimes known as "echo" effect). It is an effect which records an input signal, stores it, then plays it back after a defined time. Typically, the delayed audio is mixed with the live audio input creating an echo effect, where we first hear the original audio, followed by the delayed audio. The value of D in Equation 3.6 determines the shift of the wave, and thus also the sound. A positive value (such as

$\frac{\pi}{2}$) will shift the sine wave to the left on the Cartesian plane, resulting in a sine wave which sounds early. The same applies to a negative D value, in which a value such as $-\frac{\pi}{6}$ shifts the sine wave to the right, creating a sine wave which sounds "late" or delayed.

Other modules can be created through similar logic. To create legato and staccato, we either connect the waves of different frequencies together, or separate the waves to the point there is a clear distinction between the notes. Musically, these two concepts are total opposites; staccato is defined as the style of playing notes in a detached and separated manner, in which each note is clearly distinct from one another. It is typically indicated by a dot directly above or below the notehead, depending on if the stem of the note goes upwards (a dot is placed below the notehead), or goes downwards (a dot is placed above the notehead) [6]. An example of staccato is in Figure 3.13, in which there are dots clearly above and below some of the notes, and the location of the dot is dependent on whether the stems of the notes face upwards or downwards. Legato is the directive which defines notes to be played in a smooth and connected manner, with notes that are no longer clearly distinctive from one another, as each note will flow gracefully into the next.

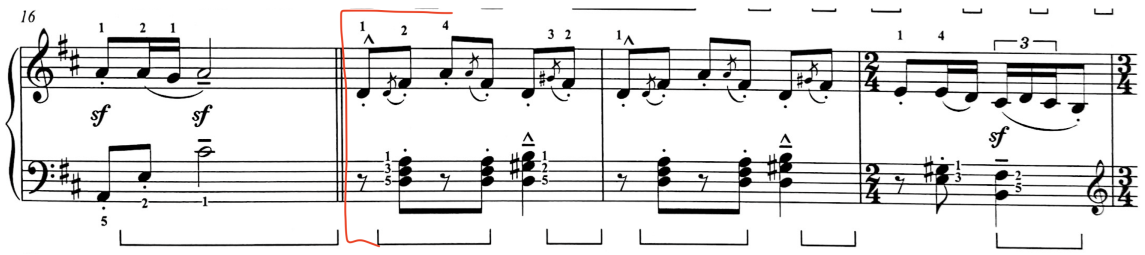

Figure 3.13: Béla Bartók, Romanian Folk Dances, *Poarga Românească*, mm. 16-19

We will also create a module meant to layer two specific harmonics over a base frequency: the major third interval, and the perfect fifth interval, which together will form a major chord (refer to Appendix C). Most chords are triadic in nature (that is, containing only three notes), with the interval of a major third or minor third between each of the three notes. The major third interval can be defined as the interval which spans four degrees of the diatonic scale in the Western twelve-semitone tuning system (refer to subsection 2.3.2), or four semitones [30].[3] The minor third interval contains one fewer degrees than the major third interval, thus having only three degrees of the diatonic scale, and so only three semitones.

For instance, with a base frequency of the note *A*, the interval between *A* and the note *C*♯ is a major third interval. This interval spans the distance of four semitones. The distance between the note *A* and the note *C* is shorter than the distance between *A* and *C*♯, resulting in few semitones between the notes. This interval is the minor third interval, as the note *C* is only three semitones away from the note *A*. A major interval will usually sound "happier" and brighter than a minor interval. The major third can give music a happy and playful tonality, while the minor third can result in music which sounds sad, dark, or spooky. It is important to understand the differences in the major and minor intervals–especially the major and minor thirds. These intervals can alter the timbre and quality of the sound (refer to Appendix C) and the feeling of the music.

Songs which contain prominent examples of the major third interval include "When the Saints Go Marching In," Ludwig van Beethoven's *Fifth Symphony*, and the spiritual song and Christian hymn "Swing Low, Sweet Chariot." In Figure 3.14, the notes must be read from the top of the sheet music down to the bottom–to understand the composite sound each instrument brings to the symphony–before being read from left-to-right. The first two measures of the symphony (before the blue line) show one major third interval, and the last two measures (after the blue line) show a different major third interval. These intervals do not sound as happy or bright as the major third interval may typically be, but the distinction between the major and minor third becomes easier to understand in comparison to

[3]The major third interval is also enharmonically equivalent to the diminished fourth interval. The enharmonic interval describes notes which sonically are the same, yet notated differently.

a well-known piece which begins with the minor third interval: the English folk song "Greensleeves."

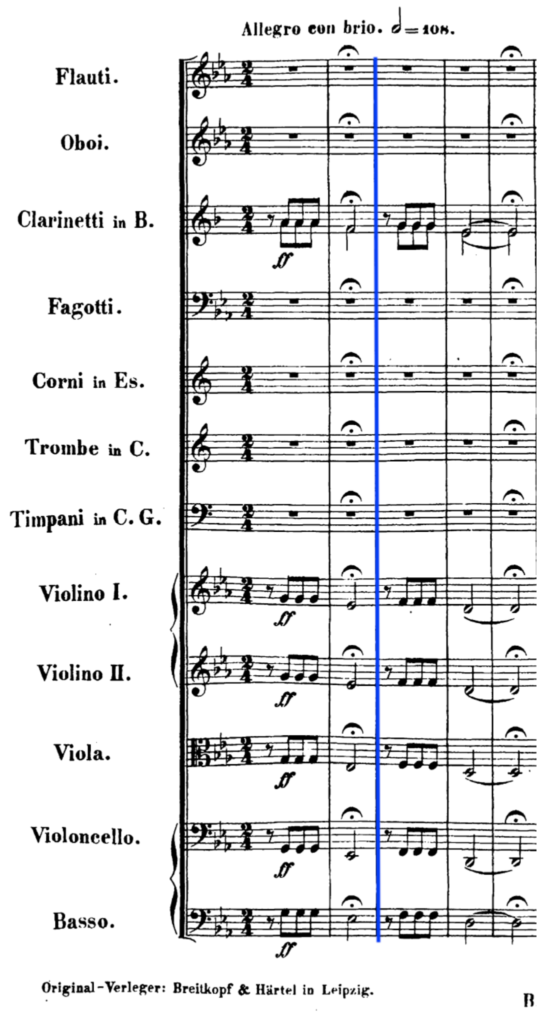

Figure 3.14: Ludwig van Beethoven, Symphony No. 5 in C Minor, *Allegro con brio*, mm. 1-4 [3]

Songs which contain the minor third interval include "Greensleeves" (the first two notes), Christmas tune "What Child Is This," and The Beatles' "Hey, Jude." In Figure 3.15, the notes must also be read from top-down, before being read left-to-right. The first two notes of the piece (as bracketed in blue) demonstrate the minor third interval. In this piece, the notes played in the two clefs (the top:

treble, the middle: alto, refer to Appendix C for more details) are different pitches, but are still minor third intervals. For instance, the two notes played in the treble clef are *A* and *C*. These notes create a sad, dark, and slightly ominous tonality to the song, a quality specific to the minor third interval. In comparison to the major third interval from Ludwig van Beethoven's *Fifth Symphony*, it is clear that there is a difference in sound quality and tone between the two interval types. The major third interval, and thus the frequency difference between a base frequency and its corresponding major third interval, will sound brighter than the minor third interval. The frequency difference between a base frequency and its corresponding minor third interval results in a sad and dark sound quality.

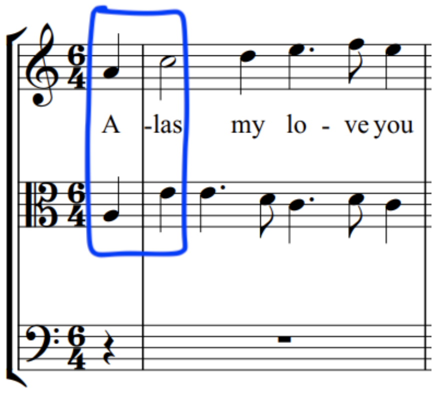

Figure 3.15: "Greensleeves", mm. 1

[23]

The interval of the third is important to distinguish *major* chords, and *minor* chords, as major chords will have a root note (the tonic note, typically the first note played of a chord), major third interval, and another minor third interval (or perfect fifth interval above the tonic) stacked on top of one another, while a minor chord will have the tonic, minor third, and perfect fifth. Either multiple waveforms (of the specified major third and perfect fifth intervals) can be stacked to produce the major chord sound, or MIDI inputs (through the use of a MIDI device such as a MIDI keyboard) can trigger a major chord.

Another manipulation possible is to add the distortion alteration (as opposed to the specific distortion effect) to a sound is simply to add desired textures (additional layers, refer to Appendix C). to a sound, through changing and deforming an audio signal's waveform. For many, a prime example is seen with the use of an electric guitar, as the pedals used with an electric guitar allow for added harmonics, and other changes made to the guitar's sound. One of the most used types of distortion is known as *clipping*, in which the level of a signal (typically amplitude) goes beyond the maximum that a system is able to handle, leading to clipping, as the maximum of the waveforms gets abruptly cut off at the system's maximum. At its best form, distortion can be a gentle audio effect, which can add many types of sounds to a signal, including saturating the sound, and adding overdrive and *fuzz*–two types of distortion effects–through adding *gain* (defined as an increase in some type of value). With respect to distortion effects, *gain* is referred to as *transmission gain*, in which there is an increase in the power of a signal, expressed in *decibels* (dB), usually done through an arbitrary combination of increasing the amplitude and frequency of a sound wave. This arbitrary combination of changes in frequency and amplitude can be done in all situations. The two most common, and most subtle, types of distortion are saturation and clipping. The result of these two types of clipping is "soft clipping" in which the peaks of the signal's waveform are softly rounded, and not abruptly cut off [44]. The signal will be pushed only slightly over the 0 dB threshold.

The concept of the 0 dB threshold is important for distortion effects, as it is a fundamental aspect to effectively create distortion within a modular synthesizer. Both digital and analog meters for volume, as in Figure 3.16, have ranges between negative infinity (or silence), up to 0 dB (the absolute loudest). In SuperCollider, this range will be between -80 and 0, as in Figure 3.16. These decibels are different than the standard decibels used to describe the loudness of everyday sounds. Standard decibels allow us to compare the relative loudness of sounds to each other; a jet taking off sounds at 140 dB, a firecracker is 140-165 dB, and a whisper may be 30 dB [14]. These decibels act as a unit measurement for sound, and the National Institute of Occupational Safety (NIOSH) [14] states that while exposure to noise at 85 decibels or above will cause hearing loss, the exposure dangers for higher levels become exponentially more damaging. While at a noise level of 70 dB would take over 24 hours to cause hearing damage, sound at a level of 115 dB would cause hearing damage at only 28 seconds. The 115 dB volume of a rock concert and symphonic orchestra concert is much more noticeable, especially when comparing to the volume of listening to music on personal devices at maximum volume (105 dB).

The type of decibels used for music production are “Decibels Full Scale” (dBFS) when discussing digital music, or “Sound Pressure Level” (dB SPL) in the real world. This is the measurement of decibels as it pertains to the levels in an audio recording. Unlike the scale for dB, in which 0 dB is absolute silence, and higher numbers indicate a louder perceived volume, the scale for dBFS is reversed. With the dBFS scale, 0 dB is the maximum level of audio a system can process before it “clips” the signal. The lowest detectable level of sound in the system, the “noise floor,” can be as low as -150 dB, but is typically between -80 dB and -90 dB. In general, the common range for volume lies between -10 and -18 dB.

So, keeping the noise level somewhere between -80 dB and -10 dB will help in introducing distortion to pure sound waveforms. If a signal is “softly clipped” (boosted slightly over the 0 dB threshold), the output sound may contain subtle harmonics (other frequencies overlaid on top of the original frequency) or other overtones.

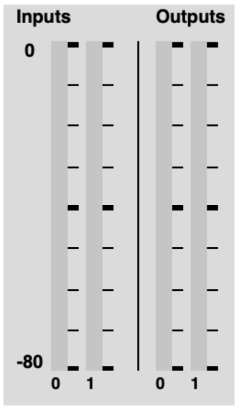

Figure 3.16: The server meter in SuperCollider

Overdrive, and fuzz–as previously mentioned–are two common types of distortion. In addition, the distortion effect itself (different from the class of sound alterations known as distortion), is another type of distortion. These three types of distortion effects tend to be synonymous with electric guitar rigs, pedals, and other similar hardware. Overdrive tends to be the most subtle of the three, with higher gain levels. Distortion and fuzz are more intense, with distortion allowing for large amounts of sustain, harmonics, and a mostly altered sound from the original input, noticeable in heavy rock music and guitar solos. Fuzz is similar to distortion in gain level, but will produce a sound similar to that of a square wave. Distortion using the fuzz effect produces a traditional synth-like effect, with digital artifacts, or overly processed sounds.

CHAPTER 4

Building a Modular Synthesizer

In the work done on creating a virtual synthesizer, the result is two input options: pure sound tones (such as the sine waves, square waves, and other waveforms previously mentioned in Section 3.1), and MIDI. Due to the continuous movement of pure sound waves, modules built for pure tones will be somewhat different from the modules which are built for MIDI input. The following section focuses on the pure sound tones, and the modules built to manipulate these waves. Then, we discuss the ways in which a user can interact with this project's synthesizer through the use of MIDI, and the various sound alterations that result.

4.1 The Modules: Pure Waveforms

The waveforms created through SuperCollider are through Unit Generators (UGens) such as the "SinOsc" Unit Generator. As in Listing 4.6, a simple sine wave is created, with a frequency of 440 Hz (A_4, defined as the A in the fourth octave of the keyboard, or Concert A), and a volume of 50. The `SinOsc` UGen defines the specific properties (frequency, amplitude) of the sine wave, as well as the wave itself. Thus, as previously seen in both Figure 3.3 and Equation 3.6, a continuous sine wave has been created.

```
SynthDef("sinewave", {arg freq=440, vol=50; Out.ar(0,
  ↪ SinOsc.ar(freq, 0, vol))}).add;
```

Listing 4.6: Creating a sine wave SynthDef in SuperCollider

4.1.1 Volume Slider

Volume is simply how loud or soft the human ear hears at a particular frequency. The sine wave SynthDef created in Listing 4.6 contains a variable value for volume. In the SynthDef, the volume is initialized at the number 50, which generally correlates to a medium-loud volume of *mezzo forte*. The variable A of the generic sine wave equation in Equation 3.6 is equivalent to this change in volume.

As the sine wave Synth Def contains two variables, one for frequency, and one for volume, creating modules for both a volume slider and a pitch knob is straightforward. To create the volume slider, a SuperCollider class called *EZSlider* creates the outline of the volume slider itself, as in Figure 4.1. For the slider's functionality, there are three important parts: the "controlSpec," "action," and "initVal." The "controlSpec" defines the "control spec," or the range of values allowed for the specified module. Negative volume does not exist, so this simple volume module will contain valid values for 0 volume (soft, or *pianissimo*) up to 100 volume (loud, or *fortissimo*). Then, the "action" argument of the *EZSlider* class determines the function that runs when the value of the volume slider is changed.

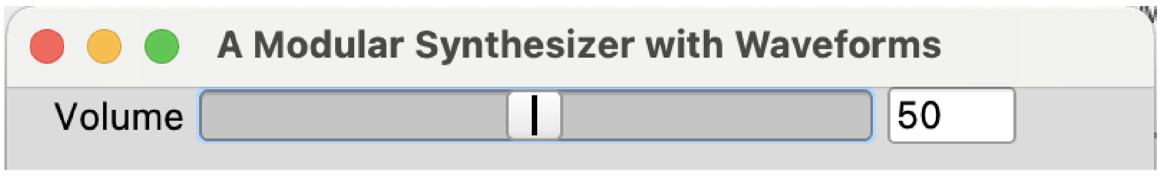

Figure 4.1: The basic volume slider, with a volume of 50, or *mezzo forte*

```
volumeSlider = EZSlider(awindow, label:"Volume",
    ↪ controlSpec:[0,100], action:{|volumeSliderValue|
    ↪ x.set("vol", volumeSliderValue.value)}, initVal:50);
```

Listing 4.7: Creating the volume slider in SuperCollider

```
x = Synth("sinewave");
```

Listing 4.8: Putting the sine wave SynthDef into a Synth, for sound output

In Listing 4.7, the action that is set involves changing the volume of the Synth created in Listing 4.8. In this code example, the SynthDef from Listing 4.6 that is assigned to the variable name "sinewave" is now put into a Synth. This Synth, as previously described in Section 2.4, allows SuperCollider to deal with audio output, as it pushes the created sine wave into the SuperCollider server *scsynth* for immediate playback. So, the SynthDef "sinewave" is put into a Synth, known as "x." Within the action, while the sine wave is wrapped in a Synth, x can be manipulated. That is, the sine wave which is in the Synth can be manipulated, resulting in an altered sound. The action itself begins with a reference to the volume slider, *volumeSliderValue*. The Synth, x, then references the volume argument from the SynthDef, "vol," which sets the volume for both the SynthDef and the Synth, and uses the `set` function to set the volume of the Synth equal to the volume that the volume slider contains. The volume of the Synth x will update as the value of the volume slider does, setting the value of `x.vol` to be equivalent to `volumeSliderValue.value`. Finally, "initVal" simply initializes the starting value of the slider to volume 50 (*mezzo forte*).

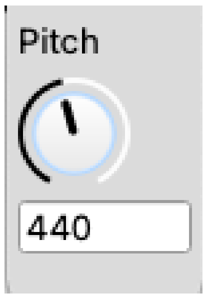

Figure 4.2: The basic pitch wheel, with a pitch of 440 Hz, or A_5

4.1.2 Pitch Knob: The Pitch Bend

Pitch, as previously mentioned, is simply a functionality which dictates the frequency of a note that the human ear perceives. Standard Western tuning currently dictates notes to be tuned around the starting pitch of the note A above Middle C (or A_5), which is equivalent to 440 Hz. We reference Listing 4.6 again, as a sine wave with both frequency and volume arguments was created here. The frequency of this sine wave is equivalent to the variable B in the generic sine wave equation (Equation 3.6).

To create the pitch knob, similar logic to that of the volume slider above is used. In SuperCollider a native class, *EZKnob*, is used. This class will create the knob, as in Figure 4.2, to adjust the frequency of the pure tones. This knob class has three arguments: the "controlSpec," "action," and "initVal." In Listing 4.9, we see that the controlSpec for the pitch knob is `freq`, denoting the span of audible frequencies to the human ear, with negative frequencies invalid. The "action" argument works similarly to its functionality in the volume slider. It will set the value of `freq` argument of the SynthDef "sinewave" equal to the frequency value of the pitch knob. The final argument of the *EZKnob* class is "initVal," in which the initial value of the pitch knob is set to 440 Hz.

```
pitchKnob = EZKnob(awindow, label:"Pitch", controlSpec:\freq,
    ↪ action:{|pitchKnobValue| x.set("freq",
    ↪ pitchKnobValue.value, currentFreq)}, initVal:440);
```

Listing 4.9: Creating the pitch knob in SuperCollider

4.1.3 Legato Switch: A Sustain Button

The *legato* switch for this synthesizer is meant to emulate the *legato* notation found in classical music. *Legato* is a directive, typically found in its full form in classical music, which indicates the performance of a specific passage to be played in a smooth, graceful, and connected style (opposed to the *staccato* notation) [47]. It will often be indicated by a slur over the notes, or an accent mark with a line over the notes to be affected, as in Figure 4.3 [17], marked in blue. Here, notes of differing pitches are to be played in a connected manner.

On a physical electronic keyboard, this module is most often seen with a *sustain* button, in which the notes played are extended, and slurred into each other. However, it is important to note that not all slur lines in written sheet music will be meant to be played *legato*. Notes which are of the same pitch cannot be played legato, as notes meant to be played legato must be of different pitches. If notes meant to be played in a connected manner are of the same pitch, a *tie* is added to these notes, as in Figure 4.4 [24] (refer to Appendix C), which connect notes that are the same pitch. Within Figure 4.4, two notes of the same pitch (circled in blue) have a line, which denotes that these notes should be played smoothly. It is important to note that these notes are of the same pitch, and so are *tied notes*, rather than notes that should be played legato. So, the durations of these notes which are tied together are combined, and played at that new longer note duration instead.

In this synthesizer, a pure sound wave was created through the SynthDef, and later put into a Synth for sound output. There is no need to develop a functionality which would connect the waveforms of one frequency to another smoothly, as pure tones will sound in a smooth transition between frequencies naturally. Thus, to develop such a module for pure sound waves is not needed. Through the nature of pure tones, the waves will move continuously unless stopped by an external force.

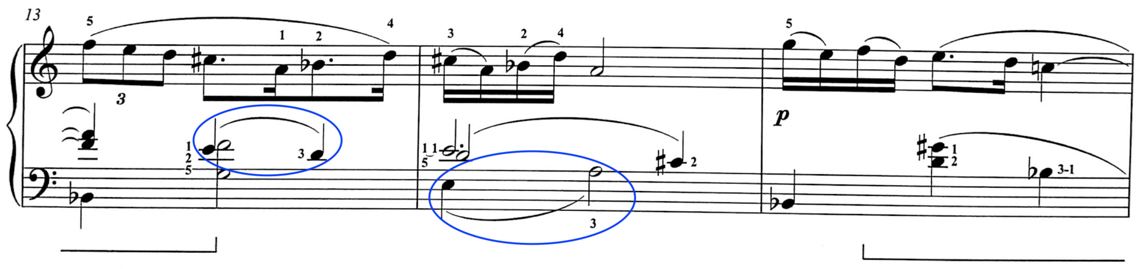

Figure 4.3: Béla Bartók, Six Romanian Folk Dances, *Buciumeana*, mm. 13-15

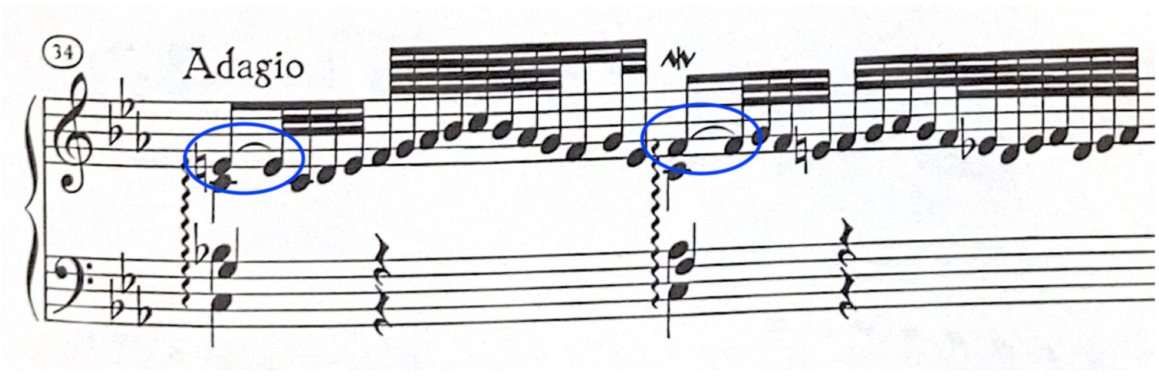

Figure 4.4: Johann Sebastian Bach, The Well-Tempered Clavier Book I, *Prelude in C Minor*, mm. 34-35

4.1.4 Major Chord Generator

The Major Chord Generator within this modular synthesizer involve two pieces: calculating the proper frequencies of the major third and perfect fifth intervals–as they are subject to change from the user's input on the pitch knob–and adding these two intervals to play simultaneously with the original waveform sound. First, to calculate the proper intervals of tonic note (the base frequency, otherwise known as the "root note") to major third, and tonic note to perfect fifth, *interval ratios* (the widths of semitones, and the calculation of frequencies relative to a base frequency, typically A440) are used (refer to Appendix C for further definitions for each of these terms). As mentioned within Subsection 2.3.2, the commonly used tuning system in the Western world is the twelve-tone equal temperament system, which divides the octave into 12 parts. Each of these parts are equally tempered (or equally spaced) on a logarithmic scale, such that each ratio is equal to $2^{\frac{1}{12}}$, or $\sqrt[12]{2} \approx 1.05946$ (the 12th square root of 2, and a in Equation 4.1 [42]). This tuning system is normally tuned relative to the standard pitch of 440 Hz, known as *A440*, signifying that the note A (typically A_4, or Concert A) is tuned to 440 Hertz, and all other notes are defined relative to this pitch, as some multiple of semitones away from it, either higher or lower in frequency. Thus, the modular synthesizer created also begins with a starting pitch of A440, every other pitch is defined relative to it (f_0 in Equation 4.1).

To calculate an interval from our base frequency of 440 Hz, we must multiply by some value $\frac{n}{12}$, with n the distance, in semitones, from the base frequency to the interval we want to find. We want to calculate the frequency value of the major third interval and the perfect interval, based on the base frequency value of 440 Hz. We know that the major third interval is four semitones away from the root note, and the perfect fifth is seven semitones away from the root note. To calculate the major third interval, like in Listing 4.1.4 in which we define the values of variables `thirdFreq` and `fifthFreq`, we multiply our base frequency by some multiple of $\sqrt[\frac{n}{12}]{2}$. As the major third interval has a distance of four semitones away from the root note, we set $n = 4$, to calculate $f_n = 440 * 2^{\frac{4}{12}}$. We do the same thing to calculate the perfect fifth interval. This interval has a distance of seven semitones from the root note, so we multiply our base frequency by $\sqrt[\frac{7}{12}]{2}$, resulting in $f_n = 440 * 2^{\frac{7}{12}}$. Through these two calculations, we understand that with a base frequency value of 440 Hz, the values of `thirdFreq` and `fifthFreq` of Listing 4.1.4 to be:

```
thirdFreq = 554.37;

fifthFreq = 659.25;
```

$$f_n = f_0 * (2)^{\frac{n}{12}} \tag{4.1}$$

After calculating the ratio of frequencies, the work on the second part of the module begins. Similar to the work done to create the initial waveform, two additional pure waveforms, and their Synth counterparts, are created (Listing 4.1.4). Then, the only step which remains involve adding the Synths y and z into a variable to play simultaneously (Listing 4.1.4) using an array called “majChord.”

```
var baseFrequency = 440;
var thirdFreq = baseFrequency * (2**(4/12));
var fifthFreq = baseFrequency * (2**(7/12));
...
SynthDef("sinewave_third", {arg vol=50; Out.ar(0,
    ↪ SinOsc.ar(thirdFreq, 0, vol))}).add;
SynthDef("sinewave_fifth", {arg vol=50; Out.ar(0,
    ↪ SinOsc.ar(fifthFreq, 0, vol))}).add;
y = Synth("sinewave_third");
z = Synth("sinewave_fifth");
...
majChord = ["sinewave", "sinewave_third", "sinewave_fifth"];
```

Listing 4.10: Creating the major third and perfect fifth intervals, and placing the three notes into an array, for playback on the SuperCollider server. The ellipses describe code which is not relevant to show for demonstration purposes.

To create the button for a clean and easy-to-use user interface, we use the native SuperCollider class, *Button*, and assign the value of this button to the variable “majorChord.” This button has two states, on and off, and the variable `majChord` of

Listing 4.1.4 is triggered when this button's state is on. Once `majChord` is triggered, all three pure tones sound at once. For demonstrative purposes, we show that two sine waves are layered on top of the sine wave SynthDef which we began with, to create a major chord.

```
majorChord = Button(awindow, Rect(20, 20, 150,
    ↪ 25)).states_([["Turn Major Chord Off", Color.black,
    ↪ Color.gray], ["Turn Major Chord On", Color.black,
    ↪ Color.yellow]]);
```

Listing 4.11: Implementing the major chord module using the *Button* class

4.1.5 Delay Slider

Adding delay to a pure tone involves affecting when a waveform will move. Also known as an "echo" effect, this effect results in a sound being perceived as earlier or later than it "should" have been played. For this module, we create a slider, using the native SuperCollider class *EZSlider*. This module will determine the time the waveform is delayed. Like with the volume slider, we are using the same three arguments: controlSpec, action, and initVal. For the control spec of the delay slider, we must make sure these values match those available in the unit circle, so that the valid range of values is between $\frac{\pi}{6}$ to $\frac{11\pi}{6}$, as in Listing 4.12. While it is possible to use values greater than $\frac{11\pi}{6}$, or less than $\frac{\pi}{6}$ in theory, in practice it will sound equivalent to values within this range, as the sine wave will be overlaid directly on top of the possible values in this range.

In SuperCollider the time values for delay are calculated in radians, so using the values from the unit circle works well. For the "action" argument, similarly to previous modules, the phase of Synth "x" is set to be equivalent to the value of the delay slider. The slider itself is initialized to 0, where there is no early or late arrival of the sound.

```
delaySlider = EZSlider(awindow, label:"Delay Time",
    ↪ labelHeight:50, labelWidth:100, controlSpec:[(-pi)/6,
    ↪ pi/6], action: {|md| x.set("phase", md.value)},
    ↪ initVal:0);
```

Listing 4.12: Creating a delay slider in SuperCollider

4.1.6 Adding Distortion

Pink noise is a type of noise which contains all the possible frequencies which a human can hear. Unlike other types of colored noise, pink noise is much less intense. There are multiple types of colored noise, including black, red, blue, brown, and white. However, for music production, the two most popular types of colored noise are white noise and pink noise. White noise operates similarly to white light, encompassing the entire frequency of audible sound from low pitches to high pitches equally. Different frequencies are played randomly across the entire audible range, and normally sounds like radio static [46], as in Figure 4.5. When mixing white noise into a pre-existing music mix, white noise fills sonic space similar to how low bass notes fill sonic space in the very low end (refer to Table A.1). Pink noise is very similar to white noise, but constructed differently. It creates equal amplitudes based on the octaves, getting softer and less abrasive-sounding as the pitch rises, as in Figure 4.5. Thus, lower frequencies are louder, and higher frequencies are easier to listen to and are softer [46]. As it technically has a fundamental frequency, it will sound much more natural than white noise, as natural sounds all have a specific fundamental frequency (defined as the lowest harmonic played).

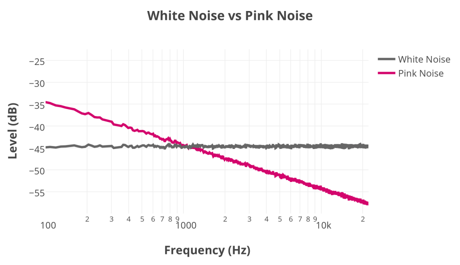

Figure 4.5: Pink vs White Noise

[13]

Due to its more natural sound than white noise, pink noise is the preferred type of colored noise to use for this synthesizer. In addition, as higher frequencies are lower in volume, and lower frequencies higher in volume, the entire frequency range of pink noise will have the same acoustic energy, since it takes a wave much less energy to play a pitch in the lower range of frequencies than in the higher range of frequencies. Then, the sound of a pure tone can be altered to be rougher, with the `baseFrequency` variable of Listing 4.13 as our starting frequency.

The distortion of a simple, pure audio signal involves a SuperCollider class known as *InsideOut*. For this module, both clipping and fuzz are used. As in Listing 4.13, there are two aspects which create the clipping and fuzz effects: a sine wave Unit Generator (UGen), and a `Pink Noise` UGen. The sine wave UGen allows for the clipping of the sound wave. The `Pink Noise` UGen is added through the fuzz effect, and allows us to add pink noise.

```
// PinkNoise function, with a sine wave UGen at the frequency
    ↪ from pitchKnob
dist = {InsideOut.ar(SinOsc.ar(baseFrequency) +
    ↪ PinkNoise.ar(0.9, 0), 30, 50)}.scope;
```

Listing 4.13: Creating a distortion module

4.2 MIDI Input

Input with MIDI in SuperCollider is more complex than creating pure sound waveforms for modular changes; however there are some similarities, we must create three aspects for modular sound synthesis. The first, the `Synth`, must be added, so sound can be sent to the SuperCollider server *scsynth* for eventual playback. This is different than sending a pure tone to *scsynth*, as sending a pure sound wave will result in the immediate playback of that wave. In working with MIDI, we do not have the issue of a MIDI note playing immediately if we were to send a `Synth` to the server, since MIDI relies on the user input through a MIDI controller. The second part, the `SynthDef`, is the way in which we define how a user-controlled MIDI input note will sound. We are using a MIDI keyboard to control this synthesizer, and so as in Listing 4.15, create a SynthDef to sound similar to a piano. Finally, we must also create the MIDI note on and note off functions for MIDI to function correctly.

```
MIDIClient.init;
MIDIIn.connect;
```

Listing 4.14: Initializing the MIDI Client

Before any of this, however, the MIDI inputs and the MIDI client must be initialized, as in Listing 4.14. These are the two key steps to using MIDI in SuperCollider: initializing the MIDI client (with the `MIDIClient.init` command) and connecting to the specific MIDI controller that will be used (through the `MIDIIn.connect` command). Once this is done, we can move on to creating the MIDI functionality itself.

The first step to creating MIDI functionality within SuperCollider will be to create the SynthDef, which will hold the instructions for how MIDI input is supposed to sound. Listing 4.15 describes the necessary aspect in the creation of a SynthDef that will be compatible with MIDI commands. The ADSR envelope, or “envelope generator" (a term which refers to the "shape" of a sound, or the contour by which a sound gets louder and softer) is described by its stages: Attack, Decay, Sustain, Release, as in Figure 4.6 [36]. Some of this was discussed in the Note On and Note Off messages subsections of Section 2.3.3. The order in which sound goes through an envelope generator is important, as sound must travel through the attack, decay, sustain, and release stages in that order, and is unable to go back to any other stage once it comes to a stage. An ADSR envelope generator will first receive a gate input, or an input signal. The gate is one of the main signal input types of a modular synthesizer, and the level of the gate input will change as it is processed through the ASDR envelope. Starting in the Attack stage, the gate input, and in Listing 4.15 will be called `gate`, is increased from its starting level (typically 0) to the level set by the Attack stage (the maximum volume the gate input will become, and the maximum volume the envelope generator is able to output). This occurs when a new note is meant to start, such as when a user presses down on a MIDI keyboard, or another transition is meant to happen, such as when the next stage of the ADSR envelope is meant to start. When a user presses down on a MIDI keyboard, the gate will typically stay at the level it was given for the duration of that note, and then suddenly drop to its baseline level once the key is released. Thus, when the `gate` variable is sent through a typical envelope generator like the ADSR envelope generator, the beginning of the gate increases to its maximum volume as it tells the envelope to go through the Attack and Decay stages. As the gate remains at this high level, the envelope may go into the Sustain stage, and then when the gate's volume returns to its baseline level, the envelope will move into the Release stage.

```
SynthDef("piano", {arg freq = 440, amp = 0.1, gate = 1;
var snd, env;
env = Env.adsr(aLevel, dLevel, sLevel, rTime, amp).kr(2, gate);
snd = Saw.ar(freq: [freq, freq*1.5], mul: env);
Out.ar(0, snd)
}).add;
```

Listing 4.15: Creating a MIDI SynthDef with an ADSR envelope

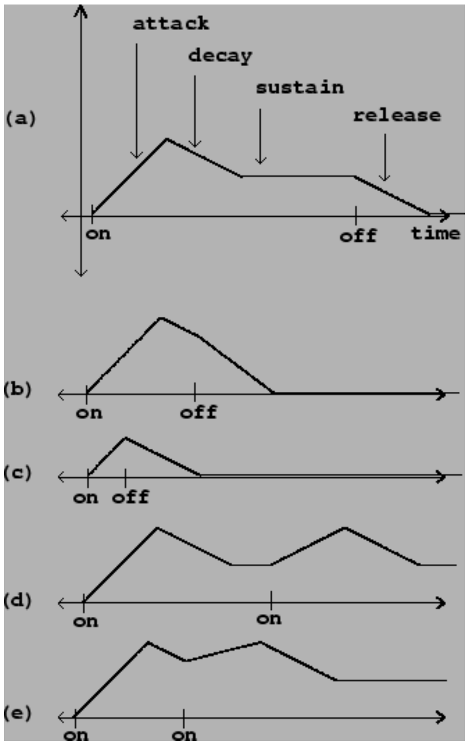

Figure 4.6: ADSR envelope output: (a) with "on" and "off" triggers separated; (b) and (c) with early "off" trigger; (d), (e) re-attacked.

[36]

Once the sound reaches this defined high level of the Attack stage, the Decay control will cause the sound to begin dropping in volume, until it reaches the volume set by the Sustain control. If the `gate` variable is still active (and has a value other than 0), the level set by the Sustain control is maintained until the value of `gate` returns to 0–which typically signifies the user has released the key on a compatible keyboard or controller. When `gate` is no longer active, the output volume begins to drop back to a volume of 0, in which the rate of this drop is determined by the Release control.

A key concept to understanding the ADSR envelope is that there is a difference in behavior when the ADSR envelope is not able to finish the entire four-stage cycle. If the user were to release the key before the Attack or Decay stages finish, then the envelope may skip to the Release stage, passing over the Sustain control entirely, and continuing to the Release control with the current volume level. However, if the user were to re-trigger the envelope by sending a new `gate` through to the synthesizer, a digital envelope generator will return the volume level back to 0, and restart the envelope cycle.

```
on = MIDIdef.noteOn(\keyDown, {arg vel, note, vol;
  notesArray[note] = Synth("piano", [\freq, note.midicps,
     ↪ \amp, vel.linlin(aLevel, dLevel, sLevel, rTime)]);
 });

off = MIDIdef.noteOff(\keyUp, {arg vel, note;
  notesArray[note].set(\gate, 0);
 });
```

Listing 4.16: Creating MIDI note on and MIDI note off messages

Other arguments within the SynthDef of Listing 4.15 include `freq`, and `amp`, which will help in modifying the input MIDI key presses. Creating the Synth and MIDI functions for proper MIDI functionality are much more difficult. When creating the waveform modules, there was no need to develop functionality for the waveforms, as they were a native aspect of SuperCollider. Within the MIDI modules, creating the Synth and MIDI functions must be done simultaneously. As in Listing 4.16, the two primary MIDI messages that will be used are note on and note off. To create a Synth using the “piano” SynthDef from Listing 2.3, it will be wrapped within the SuperCollider MIDI function for note on: `MIDIdef.noteOn`.

With the proper MIDI functionality created, similar modules to those developed for pure waveforms can be made.

4.3 The Modules: In Midi

Each of the modules built for MIDI input rely on the usage of the ASDR envelope created in Listing 4.15. Within SuperCollider, the `Env` class supports the creation of an ADSR envelope, which will allow us to create the modules for the MIDI controller. As in Listing 4.17, SuperCollider has native functionality for an ADSR envelope [27]. The first four arguments of the `ADSR` envelope are consistent as we create the maximum levels for the Attack, Decay, Sustain, and Release stages. The other variable which we are interested in is `peakLevel`, which in this project is known as `amp`. This variable is different from `gate`, as seen in Listing 4.15. The `gate` variable helps determine the current active stage of the ADSR envelope, helping sound move through it one stage at a time. The `amp` variable on the other hand defines the peak level of each of the stages itself. It is the defining high level of the Attack Stage. Through changing the levels of the Attack, Decay, Sustain, and Release stages, we can modify the input sound from a MIDI controller.

```
Env.adsr(attackTime: 0.01, decayTime: 0.3, sustainLevel: 0.5,
    ↪ releaseTime: 1.0, peakLevel: 1.0, curve: -4.0, bias: 0.0)
```

Listing 4.17: Template for creating an ADSR envelope in SuperCollider

4.3.1 Volume Slider

The first module of the MIDI synthesizer, the “volume” slider, is a misnomer. While the result is the same, and the output sound from the MIDI controller has a different dynamic level, the logic of the module is different from that of the volume slider within the waveform synthesizer. The variable amp determines the peak level of the MIDI note within the ADSR envelope, or the maximum level of the amplitude of the MIDI input. Thus, the manipulation of amp is important to adjusting the output volume heard by the user. To then create this amp slider, the SuperCollider class *EZSlider* creates the amp slider, with three arguments: "controlSpec," "action," and "initVal." The control spec for this slider is the same as the volume slider for the waveforms, and cannot be negative, as both negative volume and thus a negative amp value does not exist. Valid values of the control spec is thus between 0.1 and 1, the typical range for an amp value. Then the "action" of the slider will determine the volume output of the MIDI input.

```
ampSlider = EZSlider(awindow, label:"Amp Volume", labelHeight:
    ↪ 100, labelWidth: 150, controlSpec: ControlSpec(0.1, 1,
    ↪ \lin), action:{|ampSliderValue| note.set(\amp,
    ↪ ampSliderValue.value)}, initVal:0.3);
```

Listing 4.18: Creating the amp slider for MIDI

In Listing 4.18, the "action" involves altering the input signal from a MIDI controller. The note which holds the information for each note on message is accessed, and the value of the note's amp is set to be equivalent to the slider's value. The amp value will not continuously update itself, a difference from the volume slider. This difference in functionality is due to the nature of both pure sound waveforms and MIDI signal inputs. Within pure sound waveforms, the signal is continuous, and continues to sound until an external force stops the signal. MIDI signals, on the other hand, rely primarily on the usage of note on and note off messages to send sound to a tool which can properly output it. Thus, the amp of the MIDI note on message of Listing 4.16 will only update when a new MIDI note on message is sent. Finally, the initial value of the slider is at a *mezzo forte* value of 0.3.

4.3.2 Major Chord Generator

A major chord generator for MIDI will create a major chord, by sounding two additional frequencies on top of a MIDI note which a user plays. This will create a triadic chord–or a chord with three frequencies in total–made up of the user-input MIDI note and two intervals above the base MIDI note: the major third, and the perfect fifth. The major third interval spans four degrees of the diatonic scale within equal temperament, and the perfect fifth spans seven. As with the major chord generator which was built for pure sound waveforms, the major chord generator for MIDI involves the same two aspects: calculating the proper interval frequencies for the major third and perfect fifth interval, and pushing these two intervals to the SuperCollider server *scsynth*. Using Equation 4.1, we understand that the interval of a major third will have a frequency of $440 * 2^{\frac{4}{12}}$ and the perfect fifth a frequency of $440 * 2^{\frac{7}{12}}$.

One key difference between creating a major chord generator for pure tones and a major chord generator for MIDI input is the way in which we must push the major third interval and perfect fifth interval to the SuperCollider synth *scsynth*. When we created this module for pure sound waveforms, we created the major third and perfect fifth intervals, then put our base frequency and these two intervals into an array for playback. However, for MIDI input, the process of putting all three frequencies into an array will not be necessary. This is due to slight differences in the way in which we created the SynthDefs for the pure tones and the MIDI input. For pure sound waveforms, as we put UGens (such as *SinOsc*) into a SynthDef, then pushed it to *scsynth*, the result was that each wave was totally independent. Each pure tone did not interact with another unless an external tool (such as the array) was used to place the waves into one structure. This structure would then allow for the layered playback of each of the frequencies, and would sound a major chord. For a MIDI input, we no longer need to place each of the three frequencies (the base frequency, the major third interval, and the perfect fifth interval) into an array for playback. The SynthDef for MIDI input is contained within an ADSR envelope, and allows for each note input through MIDI to have its own independence and modularity, while also offering the option for these MIDI notes to interact natively. The ability of notes input with MIDI to interact with its predecessors and successors allow for more independence and cross-functionality between adjacent notes.

Thus, having already created the proper intervals for the module, we then must push the intervals to play on *scsynth*. This is different from creating this module for pure waveforms in that we no longer must put the major third and perfect fifth intervals into an array for playback. The SynthDef for MIDI is contained within an ADSR envelope, which has more modularity and independence than pure sound waveforms. Thus, we wrap both the variable for the major third interval (`thirdInterval`) and the perfect fifth interval (`fifthInterval`) into variables titled `thirdIntervalSynth` and `fifthIntervalSynth` respectively. These variables utilize similar logic to the implementation of `MIDIdef.noteOn`, layering the intervals on top of the user input. Both the major third interval and perfect fifth intervals will then sound once the user pressed down a key to signify the start of a note on message. With this message, the user-input note will begin going through the stages of an ADSR envelope, and the `thirdIntervalSynth` and `fifthIntervalSynth` are tied to the start of the note on message, or the state of the keypress of a MIDI controller.

```
thirdIntervalSynth = Synth("piano", [\freq, thirdInterval,
   ↪ \amp, vel.linlin(0, 127, 0, 1)]);
fifthIntervalSynth = Synth("piano", [\freq, fifthInterval,
   ↪ \amp, vel.linlin(0, 127, 0, 1)]);
```

Listing 4.19: Creating a major chord in MIDI

4.3.3 Adding Staccato

For the staccato button module, we rely heavily on two aspects of the ADSR envelope, the Attack level, and the Decay level. In combination, a shorter attack time and a shorter decay time will create a sudden perceived drop in the volume level of a MIDI input signal. As in Listing 4.16, we have created four variables which contain the values for each of the four stages of the ADSR envelope: `aLevel` for Attack, `dLevel` for Decay, `sLevel` for Sustain, and `rTime` for Release. For this module, the values for `aLevel` and `dLevel` are set to 0. With Attack set to 0, the sound of the MIDI input will hit immediately after the signal is begun. Decay set to 0 results in no time for the sound's level to fall from its peak of `amp` to the `dLevel`.

4.3.4 Adding Legato

The legato button emulates the musical concept of legato. On a physical MIDI keyboard, a legato button will more often be referenced as a *sustain* button, in which the lengths of notes are extended, and flow into one another. For the concept of legato, it is imperative that each note meant to be played legato are of different pitches (refer to Appendix C). Only notes which are of different pitches can be played legato. Otherwise, as the notes would be of the same pitch, the notes would be *tied notes*, with the composite duration of the tied notes equal to the sum of each individual note. Two other aspects of the ADSR envelope are important to extend the sounding of a MIDI note, and to connect it to another MIDI note: Attack and Release. As mentioned, increasing the level of the Attack stage will result in the sound of a MIDI input signal lengthening. This results in more time needing to pass before the gate will move onto the second stage of the ADSR envelope. Extending the time the gate is in the Release stage is also important. For proper legato within MIDI, the Release stage will need to be extended, as the ADSR envelope is only within the note on MIDI message. The note within the MIDI note on message will continue to sound, even after the MIDI note off message is sent through SuperCollider.

4.3.5 Distortion

Distortion, the altering and deforming of an audio signal's waveform, is done to add specific textures and harmonic over what an instrumentalist would play. For this module, we add regular distortion to the input MIDI signal, to add additional harmonics. Regular distortion is the easiest type of distortion to implement for MIDI, as it is similar to how distortion would be added in rock music, and guitar solos, with Jimi Hendrix songs as one example. Distortion in this module will be a kind of gentle distortion in which two pure sound waveforms are added on top of the MIDI signal, when a note on message is sent.

This module is built differently than the distortion module for pure sound waveforms. With pure sound waveforms, we used simple unit waves–such as sine waves and sawtooth waves–and altered the sound through a native SuperCollider class known as *InsideOut*. The pure sound waves were only ever able to output one frequency at a time, a limitation which does not hold true with MIDI. MIDI can output multiple frequencies at once, through the use of multiple note on messages. However, for the purposes of adding distortion, it is much easier to do with pure tones, as adding pink noise to a pure sound waveform results in a grating sound. With MIDI, though we are also adding an additional harmonic over an input MIDI signal, we are not adding colored noise.

As seen in Listing 4.20, we use the *tritone* interval to add distortion. The tritone interval is used, as instead of the pleasant sounding major third and perfect fifth intervals used in the major chord generator, the tritone sounds very unpleasant and dissonant (a sound which is not harmonious). The tritone is the interval of an augmented fourth,[1] and is made up of six semitones. For instance, if C is the base frequency, then the tritone interval away from C is F♯.

Distortion for MIDI is not meant to sound as pleasing as adding additional harmonics to an electric guitar signal through an amplifier may be. Instead, for the purposes of this module, the tritone interval is used, to give the distortion an unpleasing effect. The distortion effect is also used for this module, albeit in the simple way of adding two pure sound waveforms over the input MIDI signal. As the tritone interval is six semitones away from a root note, we first calculate the tritone interval to be $f_n = \texttt{baseFrequency} * 2^{\frac{6}{12}}$, using Concert A as the base frequency. Then, as in Listing 4.20, the tritone frequency is added to two waveform UGens, `SinOsc` and `Saw`, which will add the necessary unpleasant sounds to the MIDI input note's frequency.

```
tritone = baseFrequency * (2**(6/12));

SynthDef("distortionSynthDef", {arg out = 0;
 Out.ar(out, SinOsc.ar(tritone, 0, 50), Saw.ar(tritone, 0, 50))
}).add;
```

Listing 4.20: Adding distortion in MIDI

4.3.6 Manually Adjusting the ADSR Envelope

The final module for the MIDI synthesizer is a method of manually adjusting the values for the four pieces of the ADSR envelope. The first five modules of the MIDI synthesizer are "presets," (also known as a "patch") or pre-defined modules which are pre-programmed into the synthesizer. These presets, which are also found as all five modules in the pure sound waveform version of the synthesizer, allow a

[1]Enharmonically equivalent to the diminished fifth, both the augmented fourth interval and the diminished fifth interval will sound the same to a listener, but are notated differently.

user to understand how an output sound may be changed, without requiring a more in-depth understanding of waveforms or ADSR envelopes. Presets found within a synthesizer are typically modules built to function in a particular way, such that a certain effect can be used to a specified degree. Each synthesizer contains "parameters" (like with this modular synthesizer in its ADSR envelope) which allow a user to shape a sound, altering the sound depending on how a user may press, hold, and release keys of a controller. Thus, presets are best understood to be a "snapshot" of these parameters (the level of Attack, Decay, Sustain, and Release) at specified values.

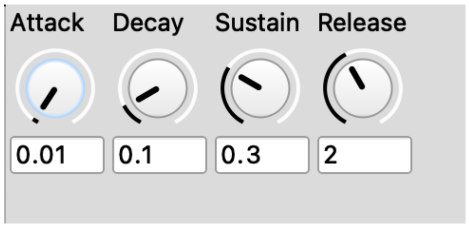

Figure 4.7: The ADSR knobs

The easiest method used to alter the levels of Attack, Decay, Sustain, and Release will be through the SuperCollider class *EZKnob*. This knob class allows for small changes in the value of a variable, and for more flexibility than a simple button may allow. This results in the ability of a user to change the sound of an ADSR envelope to their liking, and can even manually replicate the results of the five other modules, as in Figure 4.7. To create the four knobs of Figure 4.7, we use the *EZKnob* class four times, once per each knob. Each knob will have the same logic, only affecting different aspects of the ADSR envelope, and with a different initial value to the knob, as in Listing 4.21. The knob which controls Release will have a larger control spec than the three other stages of the envelope, as we have set the value of Release to be larger within the legato module, to make the legato of MIDI inputs clear.

```
attackKnob = EZKnob(awindow, label: "Attack", controlSpec: [0,
    ↪ 1], action:{|attackKnobValue| aLevel =
    ↪ attackKnobValue.value}, initVal: 0.01);

decayKnob = EZKnob(awindow, label: "Decay", controlSpec: [0,
    ↪ 1], action:{|decayKnobValue| dLevel =
    ↪ decayKnobValue.value}, initVal: 0.1);

sustainKnob = EZKnob(awindow, label: "Sustain", controlSpec:
    ↪ [0, 1], action:{|sustainKnobValue| sLevel =
    ↪ sustainKnobValue.value}, initVal: 0.3);

releaseKnob = EZKnob(awindow, label: "Release", controlSpec:
    ↪ [0, 5], action:{|releaseKnobValue| rTime =
    ↪ releaseKnobValue.value}, initVal: 2);
```

Listing 4.21: Manually adjusting the values of an ADSR envelope

As stated, the ability to manually adjust the values for the four stages of the ADSR envelope will allow for greater flexibility, over the pre-programmed values within the other modules of the synthesizer. The combinations of increasing or decreasing the four values will result in different sounds, such that different instruments will appear to have resulted from the alterations. The values for the ADSR envelope which we create in the beginning, in Listing 4.17, create a sound similar to that of a piano. A piano has a low Attack value, a short Decay time, a relatively medium length but low amp level Sustain, and a short Release time. Other instrument and sound examples, like in Figure 4.8 [43], are also possible. By setting the Attack and Decay values to be in the middle of the control spec, and Sustain and Release close to 0, a sound similar to that of a kick drum and snare drum is achieved. To create

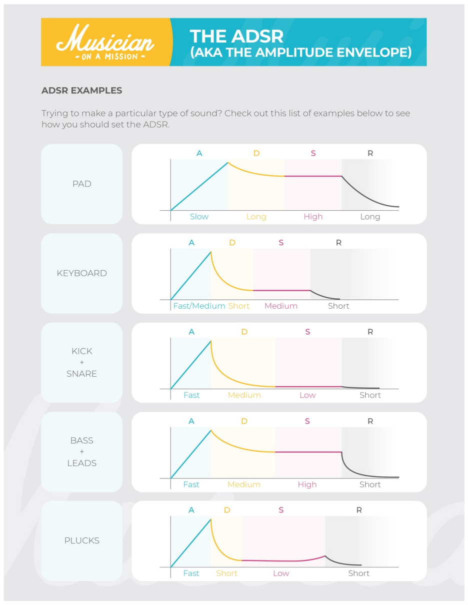

Figure 4.8: Other examples of ADSR values
[43]

a sound similar to that of a bass guitar or upright bass, set the Decay and Sustain values to be in the middle of the control spec or high, and Attack and Release values to be low. Increasing the level of Attack also will emphasize the initial "hit" of the MIDI note on message, causing the sound to appear to be closer. Within music production, this sound would appear to have emphasized the highs and mids (a frequency range of roughly 500 Hz to 4 kHz), and it would become more prominent in a mix. The same is true on the flip side, in which turning down the Attack will make the sound seem to be further away, and give the overall sound more "space" and a bigger soundstage. This is best seen in rhythm guitars and bass guitars and upright bass, in which the sound is needed, but should not be a prominent part of a mix. The other method of creating a larger soundstage within the ADSR envelope and a mix is to increase the level of Sustain, as if the microphone used to record the instrument was placed further away.

CHAPTER 5

Conclusion and Future Work

This project resulted in the successful implementation of a modular synthesizer, in which both pure sound waveforms and MIDI signals are used as inputs. Within the timeframe available, we accomplished a set of minimal goals necessary for success. While this thesis project ends with a positive result, there are also multiple ways in which this synthesizer can be improved upon, or extended. To conclude, we review the challenges of implementing this application, the necessary goals of this project, which we accomplished, in addition to the areas in which further research and development can be pursued.

5.1 Challenges

The development of this virtual modular synthesizer has achieved its goals, but limitations in the process of implementation still exist. The most noticeable in the development of this synthesizer, involves the scope of functions in SuperCollider. SuperCollider has two types of scope for functions and variables: local, and environmental (similar to how many other languages function). If a variable is declared within a certain scope, like a function, the variable will thus have a local value only within that scope, as variable a does in Listing 2.5. This code lies between parenthetical brackets (simply known as "brackets" in SuperCollider) which determine the scope of a variable. However, lowercase letters (a-z, with the exception of 's' which by default is used as a reference to the *scsynth* server) are able to be used with declaration, as are "environmental" variables, with contain the ˜ symbol [26]. While good that some variables are able to be accessed outside a particular scope, this is not good software development. Only single lowercase letters can be global, and if we were to use longer variable names, we would use environmental variables, declared with the ˜ symbol, and are seen as global variables within SuperCollider. So for the purposes of developing a modular synthesizer, we avoid the usage of global variables to store the necessary information for each module. This requires us to place each module into the same scope, nested into only one set of parenthetical brackets, rather than multiple. We lose the ability to create different functions in different files or classes of code, as the modules must maintain their modularity, or their ability to function in any order, and affect each other.

5.2 Completed Goals

There were several goals that we needed to accomplish in order to build a fully functional modular synthesizer in SuperCollider:

1. The application is capable of creating pure sound waveforms and altering these waves.
2. The application is capable of accepting and altering MIDI input.
3. The user interface is clean and simple, and makes the interactions the user has with the application easy.

It is clear that the ability of this modular synthesizer to accept and alter either type of input is a minimal requirement of this application, as it is the purpose behind the development of this synthesizer. However, that does not mean that having a clean user interface is a less-important goal. If the functionalities (modules) of this module synthesizer are not implemented with a clean user interface, then the synthesizer may be useless. It necessitates the inclusion of a high-quality and easy-to-use interface, so a user is able to understand each module within this synthesizer, as well as the ways in which each module is meant to alter an output sound.

5.3 Future Work

The current iteration of this modular synthesizer is easily able to alter input sounds and signals, and output a modification. However, this does not mean that all available features for a modular synthesizer are included in the scope of this project; there are still many ways in which additional features can be added or expanded, including:

- Implementing a pitch oscillator for MIDI, which would adjust the frequency of the input signal up or down an octave.
- Layering additional sounds and harmonics over both pure sound waveforms and a MIDI input signal, rather than solely harmonics which make up a major chord.
- Implementing various fades and filters, to clean up and remove certain frequencies from an output sound.
- Adding multichannel support.
- Increasing or decreasing the soundstage of an output sound, determining whether sounds are heard to be very close to the user, or far away.

- Adding compression, to "glue" sounds together, to make it appear that the sounds belong together, and are not simply layered over one another.

Pitch is an important aspect of both music and modular synthesizers. To obtain the correct sound for the instrument of choice, the pitch will need to be altered. For instance, if a user would like to use a bass guitar sound or a bass synth, then the sound will need to be in a lower octave, and the pitch cannot always be Concert A. The ability to layer various sounds over top of each other is an aspect of synthesis which is not fully explored in this project. Layering sounds will allow a user to create a fuller tone and sound, of which the frequencies may or may not be the same. In this synthesizer, we have layered two sounds over a user defined base frequency: a major third interval, and the perfect fifth interval. Additional frequencies which are either the same as the defined base frequency, or within a small difference of the base frequency will result in a bigger sound.

Filters are a fundamental aspect of modular frequencies, especially with the high-pass, low-pass, and band-pass filters. Each of these filter types allow a specific range of frequencies to be removed from the output sound, i.e. high-pass filters allow higher frequencies through, low-pass filters allow lower frequencies, and band-pass filters allow a band, or a certain range, of frequencies. Through the ability to filter out various frequencies, a user's desired sound output can be more easily fulfilled.

Stereo audio, or multichannel audio, is another aspect of music which is frequently used, but less often within DSP. Creating stereo sound for a modular synthesizer involves increasing the width of the sound. Most noticeable if a user uses headphones while using a synthesizer, stereo audio will feed audio into both the left and right channels, into the left and right ears separately, with slightly different sounds fed into each channel. Thus, sound can be placed as if it were to the left or right of the user, widening the sound output. Adding space to sound output from a modular synthesizer will also allow individual harmonics the room to be heard, and for a user to "locate" where the sound is coming from in space. This could also be achieved through the use of heavy reverb and delay effects. Compression would be a useful module to add as well. Using a compressor such as the *bus compressor* would compress every layer of the sound output together into one. This will make the aggregate output sound as if it were one layer, instead of multiple harmonics.

Overall, these features are not essential to create a functioning synthesizer and thus have not yet been implemented. However, these features do add additional benefits and sound modification options to the user, and would be worth implementing in the future.

APPENDIX A

The Frequency Ranges Within the Human Range of Hearing

General Frequency Range	Description of Range
<20Hz - 60Hz	The lowest threshold of human hearing. This includes many frequencies that are felt and not heard, and provides the “rumble” feeling in music. This range gives much of music its power, and is typically known as “sub-bass.”
60Hz - 250Hz	This range determines the amount of “warmth” and how full the sound is perceived to be. The notes fundamental to rhythm lives in this range, and and too much sound in this frequency range will result in the overall sound being

	"boomy," or muddy-sounding and messy. It is otherwise known as the "bass" frequency.
250Hz - 500Hz	The lower harmonics of many instruments are in this range. It is generally known as the "lower midrange" of frequencies, and can introduce listening fatigue and a telephone-quality to the sound if this range is emphasized too much.
500Hz - 2 kHz	This range is considered the middle of the midrange. It gives many instruments a sense of prominence in a mix, and determines how audible one instrument or vocalist is in comparison to another. If this range is emphasized, audio output may sound tinny and small, which could lead to listening and ear fatigue, as the human ear is sensitive to the human voice, and the frequencies it covers.
2 kHz - 4 kHz	The "upper midrange" is

	responsible for much of the attack sounds on percussive and rhythmic instruments. This range may add presence to the mix if boosted, but if it is emphasized too much, it may mask some speech recognition sounds. Listening fatigue may also set in if this range is emphasized too much, as the slightest boost in this range will result in a noticeable change in the sound's timbre.
4 kHz - 6 kHz	This range is known as the "presence" range. It defines a sound's clarity and the definition of voices and instruments that are present. If this range is boosted, instruments and voices may sound physically closer to the listener, and vice versa, with reducing this range, causing instruments and voices to sound further away. However, if this range is emphasized too much, a harsh,

	irritating sound may occur.
6 kHz - 20 kHz	This range controls the “brilliance” and clarity of sounds within the mix. Instead of pitches, this range is composed entirely of harmonics, and brings “sparkle” to the sound. This range also may easily cause ear fatigue, as an over-emphasis can increase the hiss heard, and produce sibilance, or an unpleasant tonal harshness, which can happen with consonant syllables (most noticeably: S, T, and Z), especially on vocals.

[42][48]

Table A.1: The general frequency range, within the range of human hearing

APPENDIX B

Variables Used

The following table is the variables which are commonly used through this paper.

Variable	Definition
f	temporal frequency (measured in Hertz): the number of oscillations per unit time
t	time
T	period of a wave
ω	angular frequency: frequency of a sine or cosine wave as it moves counterclockwise around the unit circle
x	position of a wave along the x-axis
y	displacement of a wave laterally
y_m	amplitude
λ	wavelength

Table B.1: Variables used in this paper

APPENDIX C

Music Theory Terms

The following list contains the definitions of frequently used musical terms in this paper. Additionally, terms which may augment the understanding of the changes made musically are included.

- **Frequency**: the perceived pitch of a sound.
- **Volume**: the perceived loudness of a sound.
- **Timbre**: the quality of a sound, which helps to differentiate between instruments.
- **Staccato**: a directive for notes to be played detached and separated.
- **Legato**: a directive for notes to be played smoothly and connected.
- **Tie/tied notes**: for this directive, a curved line is drawn over or under the heads of notes of the same pitch. This indicates that there should be no break in the playing of these notes, and should be played as one singular note.
- **Chord**: the simultaneous sounding of two or more notes. Typically, a chord will be composed of three notes in total, created a *triad*.
- **Major chord**: a chord composed of a root note (the tonic note), a major third interval above the tonic note, and a perfect fifth interval above the tonic note.

- **Tonic note**: the root note of a chord or a song, which determines the key signature.
- **Key signature**: a set of sharp (♯), or flat (♭) symbols placed on the staff at the beginning of sheet music or a section of music.
- **Major third interval**: the interval that spans four semitones. For example, the interval between *C* and *E* is a major third.
- **Minor third interval**: the interval that spans three semitones. For instance, the interval between *A* and *C* is a minor third.
- **Texture**: how a sound is organized, and the number of layers within a sound.
- **Treble clef**: a type of musical notation to indicate the pitches represented by the lines and spaces on sheet music. Also known as the "G-clef," the second line from the bottom represents the note *G* above Middle C. This clef is the most common clef seen. Typically, the treble clef will contain the note Middle C, as well as notes above Middle C.
- **Alto clef**: a type of musical notation to indicate the pitches represented by the lines and spaces on sheet music. This clef is also known as the "C-clef" or the "Viola clef," as only certain instruments, which include the viola, use this clef. The middle line of this clef represents the note Middle C.
- **Pianissimo**: a directive to perform an indicated passage of a composition or piece very softly. Abbreviated as *pp*.
- **Fortissimo**: a directive to perform an indicated passage of a composition or piece very loudly. Abbreviated as *ff*.
- **Interval ratio**: the ratios of the frequencies of pitches in a musical interval. As an interval is the "distance" between two pitches, the ratio assists musicians

to work with relative pitch measures applicable to a range of instruments intuitively, rather than a set of memorized frequency values. A simpler ratio will sound more pleasing to the ear, and thus more consonant, than complex ratios.

For instance, suppose we have a guitar as in Figure C.1. The *interval ratio* will then be inverse to the length of the string. The total length of the string in red has a 1:1 ratio, and the remaining pitches can be described as some ratio to this total string length. On the E string of this guitar (the top string of Figure C.1), the note an octave above the note E is still E. Then, this note E one octave above the root note E is 12 frets above the root. As noted in the Figure, pressing down on fret 12 of the E string (or any string) results in the length of the string being halved, and an interval ratio of $\frac{1}{2}$. This produces the note one octave above the starting note.

- **Equal temperament**: a system of tuning the scale, in which the octave is evenly divided into 12 equal semitones. It is based on the cycle of 12 identical fifths, or the "circle of fifths" [7].

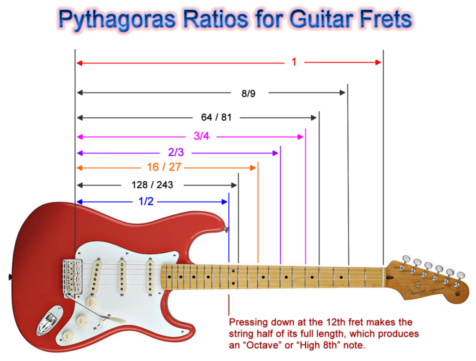
Pythagoras Ratios for Guitar Frets
1
8/9
64 / 81
3/4
2/3
16 / 27
128 / 243
1/2
Pressing down at the 12th fret makes the string half of its full length, which produces an "Octave" or "High 8th" note.

Figure C.1: Pythagoras Ratios for Guitar Frets

[32]

- **Tritone interval**: the most dissonant interval in the diatonic scale. This interval spans six semitones. For instance, the interval between C and F♯ is a tritone.

  In the Medieval era of music, the tritone was said to be the "devil's interval" because it was the most dissonant (unpleasant) interval in the diatonic scale. Its unpleasant sound to the human ear can be traced back to a phenomenon found within the human brain. The human brain is hardwired to find harmony and symmetry within music, and so enjoys consonant, pleasing sounds, and is resistant towards the less-pleasing dissonant sounds.

  Intervals which sound pleasing to the human ear are those in which there is a simple ratio between intervals (and thus also frequencies). As described in the definition of **interval ratio**, certain interval ratios are designed to sound more pleasing than others. Each note of the scale, from a root note, contains a particular frequency interval ratio. This frequency ratio, as we ascend the scale, oscillates between consonant and dissonant intervals. The tritone, as the most dissonant of all intervals on the diatonic scale, contains a large frequency interval ratio of 45:32 (or 64:45, depending on the tuning method).

- **Diatonic scale**: the scale which we have frequently discussed in this paper. It is the scale which contains, from a root tonic note, five whole tones and two semitones. Both the typically used major and natural minor scales are diatonic, with the two semitones falling between the third and fourth tones, as well as the seventh and eighth tones in the major scale. The natural minor scale is slightly different, as the semitones will fall between the third and fourth tones, and the fifth and sixth tones. From a root tonic note, each of these tones a whole tone and a semitone above can also be described in "scale degrees" (which has the same meaning as whole tone and semitone). Both the major and natural minor scales help to create the key signature, which is part of the foundation of music.

For example, in the key of C Major, the two semitone intervals will fall between the third and fourth scale degree, as well as the seventh and eighth scale degree. This results in a semitone difference between the notes E and F, and B and C. In the key of A Minor, the semitone intervals fall between the third and fourth scale degrees (C and D), and the fifth and sixth scale degrees (E and F).

References

1. Anderson, S. D. Analog and digital. 46

2. Au-Yeung, J. *The Music Producer's Ultimate Guide to FL Studio 20*. Packt Publishing, Feb 2021. 45, 51

3. Beethoven, L. v. No. 5 fünfte symphonie. op. 67. 1862. 57

4. Bode, H. History of electronic sound modification. *Journal of the Audio Engineering Society 32*, 10 (1984), 730–739. 9

5. Broughton, S. A., and Bryan, K. M. *Discrete Fourier Analysis and Wavelets: Applications to Signal and Image Processing*. Wiley-Interscience, Nov 2008. 31

6. Burkholder, J. P., Grout, D. J., and Palisca, C. V. *A History of Western Music*. W.W. Norton & Company, 2014. 24, 55

7. Cochrane, L. C. *equal temperament*. Oxford University Press, Jan 2011. 103

8. Crab, S. *Bode Audio System Synthesizer*. 2019. 5, 6

9. Crab, S. *Harold Bode demonstrating the Audio System Synthesizer*. 2019. 8

10. Dowsett, P. *Audio Production Tips*. Taylor Francis Group, 2016. 41, 43, 44

11. Drabkin, W., and Lindley, M. Semitone. *Grove Music Online* (2001). 19

12. Fleming, S. J. A. *The Thermionic Valve and Its Developments in Radio-telegraphy and Telephony*. Wireless Press, Limited, 1919. 7

13. Focus, H. Pink noise, 2022. 73

14. Foundation, H. H. Decibel levels. 60

15. Gabrielli, L. *Developing Virtual Synthesizers with VCV Rack*. Routledge, 2020. 6, 8, 9, 10, 12, 14, 15

16. Halliday, D., Resnick, R., and Walker, J. *Fundamentals of Physics*. John Wiley & Sons, Inc., 2005. 32, 33, 34, 35, 36

17. Henle, G. Das wohltemperierte klavier teil i. Jan 2009. 67

18. Huber, D. M. *The MIDI Manual, 3rd Edition*. Aug 2012. 16, 17, 22, 24, 25, 26

19. Huber, D. M., and Runstein, R. E. *Modern Recording Techniques*, 9 ed. 2018. 49, 51

20. Kapur, A., Cook, P., Salazar, S., and Wang, G. *Programming for Musicians and Digital Artists*. Manning Publications Co., 2015. 43

21. Kirk, R., and Hunt, A. *Digital Sound Processing for Music and Multimedia*, 2 ed. 2013. 17, 18, 19, 20, 23, 52

22. Krash. *Minimoog synthesizer*. Dec 2005. 6, 9

23. Kurtz, T. Greensleeves. 2010. 58

24. Lung, S. Romanian folk dances. 2016. 67

25. McCartney, J. Rethinking the computer music language: Supercollider. *Computer Music Journal 26*, 4 (2002), 61–68. 26, 27

26. McCartney, J. *The SuperCollider Book*. 2016. 90

27. McCartney, J. Supercollider 3.12.0 help, Sep 2021. 27, 28, 29, 78

28. McGuire, S. *Modern MIDI*. Taylor & Francis Group, 2014. 24, 25

29. Meyer, C. Modular synthesizer glossary of terms | learning modular, Dec 2016. 6, 9, 10, 23

30. Nave, C. Musical scales and intervals, 2017. 56

31. Nielsen, H. Elements of the west coast approach to sound synthesis. 53. 11

32. Passy. Guitar mathematics, Sep 2012. 103

33. Pinch, T., and Trocco, F. The social construction of the early electronic music synthesizer. *Icon* (1998), 9–31. 6, 10

34. Pinch, T., and Trocco, F. *Hard-Wired—the Minimoog*. Harvard University Press, 2002, p. 214–236. 9

35. Pinch, T., and Trocco, F. *Analog Days: the Invention and Impact of the Moog Synthesizer*. Harvard University Press, 2004. 6, 9

36. Puckette, M. *The Theory and Technique of Electronic Music*. World Scientific, May 2007. 75, 76

37. Romano, W. An intro to midi. *Poptronics 4*, 1 (Jan 2003), 24. 16, 19, 20

38. Rosen, S., and Howell, P. *Signals and Systems for Speech and Hearing*. BRILL, 2011. Google-Books-ID: tG6NUVhSSIIC. 1, 12

39. Russ, M. *Sound Synthesis and Sampling, 2nd Edition*. Nov 2012. 47

40. Says, T. The 'sound processor' or 'audio system synthesiser' harald bode, usa, 1959, Jan 2014. 6

41. Staff, W. C. Waves, 2021. 41, 44

42. Suits, B. Frequencies of musical notes, a4 = 440 hz, 1998. 69, 98

43. Swisher, D. Adsr: The best kept secret of pro music producers!, Nov 2019. 86, 87

44. Tarr, E. *Hack Audio*. Taylor & Francis Group, 2019. 42, 43, 59

45. Toft, R. *Recording Classical Music*. Taylor & Francis Group, 2020. 45, 46, 53

46. Unison. Pink noise vs. white noise: What's the difference, Aug 2021. 72

47. Winer, E. *The Audio Expert: Everything You Need to Know about Audio*, 2 ed. Routledge, 2018. 10, 14, 15, 31, 67

48. Zjalic, J. *Digital Audio Forensics Fundamentals: from Capture to Courtroom*. Routledge, 2021. 47, 48, 49, 50, 52, 98